Christopher Rouff · Mike Hinchey
Editors

Experience from the DARPA Urban Challenge

Springer

Experience from the DARPA Urban Challenge

Christopher Rouff · Mike Hinchey
Editors

Experience from the DARPA Urban Challenge

Springer

Dr. Christopher Rouff
Lockheed Martin
Arlington
VA 22203
USA
e-mail: christopher.rouff@lmco.com

Prof. Mike Hinchey
Lero–The Irish Software Engineering Research Center
International Science Centre
University of Limerick
Limerick
Ireland
e-mail: mike.hinchey@usa.net

ISBN 978-0-85729-771-6 e-ISBN 978-0-85729-772-3
DOI 10.1007/978-0-85729-772-3
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2011936514

© Springer-Verlag London Limited 2012
Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.
The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.
The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Cover design: eStudio Calamar S.L.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

The 3rd DARPA Grand Challenge also known as the "Urban Challenge" turned out to be the most complex of the Grand Challenges. The previous challenges were structured such that interaction with the other vehicles was minimal. We were able to test them separately. We also were able to run them on the course so that interactions, if any, occurred only when one vehicle was passing another. Even that was done under control by the chase vehicles and the Command and Control center.

The Urban Challenge however was not to only test the vehicles ability to make its way around the course missing obstacles and doing so at speed, the vehicles had to deliberately and intimately interact with other vehicles driven by human drivers but also other autonomous vehicles. We were fortunate to be able to use the housing area of George Air Force base in Victorville, California which had stop signs, alley ways and shopping centers with parking lots.

We decided to add a twist to this event and use the California Driver's test as a criterion that had to be passed. In other words, the vehicles had to show that they could pass a driving test and receive a license. This further complicated the Challenge since now we needed people out on the course writing tickets for possible infractions, greatly increasing the danger of the event.

Because of this, we had to devise qualification tests which not only tested whether or not the vehicles could navigate, but also whether or not the vehicles recognized each other and dangerous situations. This book provides good insights into the testing.

These issues and worries created a more sophisticated level of qualification testing, resulting in only 11 vehicles deemed safe enough to be let loose in the George Air Base along with human driven vehicle and people acting as traffic police.

Even though the qualification testing was rigorous, we really did not know what was going to happen the first time robotic vehicles met other robotic vehicles where they had to figure out on their own what to do. Within minutes after the start, the first interaction occurred when four of the robotic vehicles came to a 4-way stop at the same time along with their chase vehicles and other vehicles driven by humans.

The four-way stop turned out to be anticlimactic. California driving laws state that the vehicle that arrives first has the right of way and even what to do if vehicles arrived perfectly simultaneously. The robotic vehicles had no problem figuring this out and they performed perfectly. In fact, we were having problems with the vehicles driven by humans who did not always obey the rules!

It was a spectacular finish and 6 of the 11 vehicles finished the course. This book will go into greater depth on the details.

We had done what we wanted to do by showing that robotic vehicles could perform even when mixed in with other robotic vehicles and vehicles driven by humans.

I have always believed that this was DARPA's role, to show that what was thought impossible could be done. Why? Once you show that something can be done, it is amazing what progress can be made once people know that something is possible and apply resources not otherwise available.

The response from the world since the event has been fantastic. We have Driverless Taxi cabs being developed in Germany. We have Google developing driverless cars that will use GoogleMaps data. The military, for whom the research was done, is also creating convoys that have a large percentage of driverless vehicles.

In fact, if you "google" driverless vehicles, you will be amazed at the large response. In fact, as of this date, you would receive 145,000 responses highlighted by a Wikipedia article on "Driverless Vehicles."

The Challenge series may not have been the most important capability developed while I was DARPA Director, but it is very high on the list. This book will give you more details and insights into what happened at George Air Force base.

I know you will enjoy reading it even if you happened to be lucky enough to have been there.

Anthony "Tony" Tether
DARPA Director, 2001–2009

Preface

The DARPA Grand Challenge autonomous robot races that were conducted between 2004 and 2007 were initiated to foster research in the area of autonomous vehicles. While the first two challenges in 2004 and 2005 dealt with driving in unknown rural and rough environments, the challenge in 2007 increased the overall complexity by requiring correct and safe handling in an urban environment with other manned and unmanned vehicles. There was no winner in the first competition held in 2004 but the second competition in 2005 had five vehicles that finished the entire course with four of those staying under the ten hour time limit. The vehicle named "Stanley", which was developed at the Stanford University, was the winner. After a period of less than two years, the DARPA Urban Challenge was announced which included dealing with real urban-like traffic. This race had six autonomous vehicles initially from 89 applicants which finished the course within the time-limit.

This book is based on papers that originally appeared in two special sections of the AIAA Journal of Aerospace Computing, Information, and Communication (JACIC). In the chapters, different aspects of the development of the autonomous vehicles that competed in the Urban Challenge are presented. Each chapter discusses one specific aspect ranging from system overview, sensor setup, data collection, pre-processing, and data fusion, up to data evaluation and interpretation to derive a driving decision for calculating the necessary set points for control algorithms. Furthermore, aspects for system and software development processes are presented and discussed to support the quality assured development of an autonomous system under a very tight schedule. The book is divided into five parts: Introduction, System and Software Architectures of Vehicles, Navigation, Sensors and Control and Development, and Test.

In Part I of the book a brief history of the DARPA Grand Challenges is given as well as an overview of the Urban Challenge. It provides the reader with a brief background of how the Urban Challenge was structured, the two tracks of the teams, site visits, safety of the vehicles, data files that were used and the layout of the course. It finishes up with a description of the National Qualification Event and the Final Event.

Part II discusses the system and software architectures of the vehicles entered by the Tartan Racing team, the Sting Racing team, and Team Gator Nation. The Tartan Racing team chapter describes the software infrastructure developed to support the perception, planning, behavior generation, and other artificial intelligence components of Boss. It also discusses the organizing principles of the infrastructure, as well as details of the operator interface, interprocess communications, data logging, system configuration, process management, and task framework, as well as the requirements that led to the design. The Sting Racing team chapter describes their modular control architecture for autonomous navigation for urban environments that allows switching between various, and sometimes conflicting, control tasks. They describe a nested hybrid automata and illustrate its operation through experimental results. The Team Gator Nation chapter describes the control architecture of their entry which integrated the planning, perception, decision-making, and control elements. The chapter concludes with lessons learned from the Urban Challenge event.

Part III discusses how navigation was implemented by TeamUCF and Brigham Young University. The TeamUCF chapter discusses their control architecture for addressing interaction with other vehicles, such as at intersections. This architecture includes a framework that provided a means for situation assessment, behavior mode evaluation, and behavior selection and execution which is described. The Brigham Young University chapter discusses a new method of generating clothoid-based trajectories using constructive polylines. The motion planner constructs smooth, natural, drivable trajectories, and generates these trajectories using constructive polylines to create a path that is composed of a combination of straight line segments, clothoid segments, and circular arcs.

Part IV discusses sensors and the control of two vehicles from Team Cajun-Bot and Team CarOLO from the Technische Universität Braunschweig in Germany. The Team CajunBot chapter describes the steering controller of the CajunBot II, a modified Jeep Wrangler Rubicon. The CajunBot II steering controller was designed based on the look-ahead reference systems; a predictive control strategy that uses information about the heading angle and lateral displacement from the lane reference ahead of the vehicle. It's dynamic performance is also discussed. The Braunschweig University chapter presents the architecture, sensor, and data fusion concept as well as the guidance system of their vehicle Caroline. Their perception system is based on a hybrid-fusion system, combining classical object-based tracking and grid-based data fusion for obstacle detection and roadway analysis.

Part V discusses the development and test of the Team CarOLO and the Sydney-Berkeley Driving Team vehicles. The Team CarOLO chapter discusses the software engineering approach used for the CarOLO team. The chapter discusses elements of their software and systems engineering process to develop an artificial intelligence-based autonomous system capable of driving in complex urban situations. The process includes agile concepts, like a test-first approach, continuous integration of all software modules, and a reliable release and configuration management assisted by software tools in integrated development

environments, and supported by a simulate first approach. The Sydney-BerkeleyDriving Team chapter discusses their approach to urban driving with GPS outage. They give the analysis of the performance of their vehicle while navigating a test loop with severe GPS outage.

We hope that this book will not only give the reader useful background information on the Urban Challenge and some of the technologies used to develop entries to the race, but also impart insights into technology gaps that exist in autonomous systems and help develop the next generation of autonomous systems.

Christopher Rouff
Mike Hinchey

Contents

Part I Introduction

Introduction 3
Christian Berger, Mike Hinchey and Christopher Rouff

Part II System and Software Architectures

Software Infrastructure for a Mobile Robot 19
Matthew McNaughton, Christopher R. Baker, Tugrul Galatali, Bryan Salesky, Christopher Urmson and Jason Ziglar

The Sting Racing Team's Entry to the Urban Challenge 43
Matthew Powers, Dave Wooden, Magnus Egerstedt, Henrik Christensen and Tucker Balch

Development of the NaviGator for the 2007 DARPA Urban Challenge 67
Carl Crane, David Armstrong, Antonio Arroyo, Antoin Baker, Doug Dankel, Greg Garcia, Nicholas Johnson, Jaesang Lee, Shannon Ridgeway, Eric Schwartz, Eric Thorn, Steve Velat and Ji Hyun Yoon

Part III Navigation

The *V*ictor*T*ango Architecture for Autonomous Navigation in the DARPA Urban Challenge 93
Patrick Currier, Jesse Hurdus, Andrew Bacha, Ruel Faruque, Stephen Cacciola, Cheryl Bauman, Peter King, Chris Terwelp, Charles Reinholtz, Alfred Wicks and Dennis Hong

Real-time Obstacles Avoidance for Vehicles in the Urban Grand Challenge 133
Jian Yang, Jing Wang, Mi Dong and Zhihua Qu

Planning Continuous Curvature Paths Using Constructive Polylines 157
Joshua Henrie and Doran Wilde

Part IV Control and Sensors

Steering Controller of the CajunBot-II 197
Afef Fekih, Suresh Golconda, John Herpin and Arun Lakhotia

Vehicle Architecture and Robotic Perception for Autonomous Driving in Urban Environments 209
Jan Effertz and Jörn Marten Wille

Part V Development and Test

Engineering Autonomous Driving Software 243
Christian Berger and Bernhard Rumpe

Empirical Evaluation of an Autonomous Vehicle in an Urban Environment 273
Ben Upcroft, Alexei Makarenko, Alex Brooks, Michael Moser, Alen Alempijevic, Ashod Donikian, Jonathan Sprinkle, William Uther and Robert Fitch

Appendix A: Route Network Definition Files 303

Appendix B: Mission Data File 307

Index 311

Acronyms

AEVIT	Advanced electronic vehicle interface technology
ADTF	Automotive data and time triggered framework
AI	Artificial intelligence
ARC	Australian research council
ASG	Abstract syntax graph
ASM	Action selection mechanisms
AuRA	Autonomous robot architecture
CAN	Controller area network
CARMEN	Carnegie mellon robot navigation toolkit
CBSE	Component-based software engineering
CODGER	Communication database with geometric reasoning
CORBA	Common object request broker architecture
COTS	Commercial off-the-shelf
CPU	Central processing unit
DAMN	Distributed architecture for mobile navigation
DARPA	Defense advanced research projects agency
DGC	DARPA grand challenge
DGNSS	Differential global navigation satellite system
DMCP	Dynamic module configuration protocol
DSL	Domain specific language
DUC	DARPA urban challenge
ECU	Electronic control units
ECU	Engine control unit
EMC	Electronic mobility control
FC5	Fedora core 5
FOV	Field of view
FSA	Finite state automata
FSVS	Fast system verSioning
GOLD	Generic obstacle and lane detection
GPOS	Global position
GPS	Global positioning system

GPSTC	Georgia public safety training center
HDOP	Horizontal dilution of precision
HIL	Hardware-in-the-loop
HIL	Human-in-the-loop
HLP	High-level planner
HSM	Hierarchical state machine
IDE	Integrated development environment
IMU	Inertial measurement unit
INS	Inertial navigation system
IPC	Inter-process communications
IPM	Inverse perspective mapping
IPT	InterProcess communications toolkit
JAUS	Joint architecture for unmanned systems
LADAR	Laser detection and ranging
LAGR	Learning applied to ground robots
LCASS	Lane correction arbiter smart sensor
LDW	Lane departure warning
LFSS	Lane-finder smart sensor
LIDAR	Light detection and ranging
LPFSS	LADAR-based path-finder smart sensor
LWM	Local world model
MTVR	Marine medium tactical vehicle replacement
MDF	Mission data files
MO	Moving obstacles
MPEG	Moving picture experts group
MPI	Message passing interface
NASA	National aeronautics and space administration
NDDS	Network data distribution service
NFM	North finding module
NICTA	National ICT Australia
NML	Neutral messaging language
NQE	National qualification event
OSCAR	Operational software components for advanced robotics
PD	Primitive driver
PID	Proportional-integral-derivative
PRA	Point reduction algorithm
RAM	Random access memory
RCS	Real-time control systems
RFS	Richmond field station
RHC	Receding horizon control
RMS	Root mean square
RNDF	Route network definition file
RRT	Rapidly-exploring random trees
RTC	Real-time communications
RTI	Real-time innovations

RTT	Round trip time
SAI	Society of automotive engineers
SAIC	Science applications international corporation
SBDT	Sydney-Berkeley driving team
SFX	Sensor fusion effects
SLAM	Simultaneous localization and mapping
SPAN	Synchronized position attitude navigation
SRI	Stanford research institute
SwRI	Southwest research institute
TCA	Task control architecture
TCP	Transmission control protocol
TROCS	Tartan racing operator control system
TRUCS	Tartan racing urban challenge system
UCF	Urban challenge final event
UDP	User datagram protocol
UML	Universal modeling language
UPI	Unmanned ground combat vehicle perceptor integration
UPS	Uninterruptable power supply
VPFSS	Vision-based path-finder smart sensor
WAAS	Wide area augmentation system
WLAN	Wireless local area network
XSLT	Extensible stylesheet language transformations

Part I
Introduction

Introduction

Christian Berger, Mike Hinchey and Christopher Rouff

Abstract The DARPA grand challenges were conducted between 2004 and 2007 and were initiated to foster research in the area of autonomous vehicles. While the first two challenges in 2004 and 2005 dealt with driving in unknown and rough environments without any moving obstacles, the challenge in 2007 increased the overall complexity by requiring correct and safe handling of moving vehicles from other competing autonomous vehicles in an urban setting. The first competition held in 2004 afforded no winner, while the repeated competition in 2005 produced five vehicles which finished the entire course from which four stayed under the ten hour time limit; "Stanley" the vehicle which was developed at Stanford University was the winner of this first international competition. After a period of less than two years, the community achieved the increased requirements for dealing with real urban-like traffic and afforded six autonomous vehicles from 89 applicants which finished the course within the timelimit. This chapter gives a short overview of the history of the Grand Challenges, discusses the background of the Urban Challenge, as well as goes over high level rules and the qualifying process. The remaining chapters in the book cover aspects of the development of autonomous vehicles ranging from appropriate sensor technology and setup, over reliable detection of the vehicle's surroundings, up to evaluating the environment for

C. Berger
Department of Software Engineering, RWTH Aachen University, Ahornstraße 55, Aachen, Germany
e-mail: mail@christianberger.net

M. Hinchey
Lero, The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
e-mail: mike.hinchey@lero.ie

C. Rouff (✉)
Lockheed Martin, Arlington, VA, USA
e-mail: christopher.rouff@lmco.com

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_1, © Springer-Verlag London Limited 2012

deriving a driving decision are covered in the following chapters. Besides, proper software and system development methods are discussed as well.

1 Introduction

The Urban Challenge, organized by the Defense Advanced Research Projects Agency (DARPA), was the third in a series of three races to foster research and development of autonomous vehicle technology for military missions. The goal for developing the robotic technologies is to reduce the number of warfighters operating in hazardous conditions and was in support of the government mandate that one-third of the military's vehicles be autonomous by 2015. It was conducted in a competitive format to leverage the technologies from a wide range of researchers from the international community and was the first autonomous robot race of its kind.

Each of the challenges used a series of qualifying steps that led up to a final event with competitors being eliminated at each stage of the qualifying process. The qualifying steps consisted of paper submissions that covered architectures and designs, videos of the vehicles, site visits by DARPA personnel that inspected the vehicles and ran them through short courses to test their level of readiness, capabilities, and safety, a qualifying event where vehicles went through a small version of the main course that tested a range of obstacles and road conditions similar to what would be on the final event, and then the final event that was the race. This process allowed a wide number of initial applicants while narrowing the field to the most promising entries at each step. There was international participation from Germany, Australia, and from a high school, universities, industry and private individuals.

2 History of the DARPA Grand Challenges

There have been three Grand Challenges sponsored by DARPA. The first one was a race through a desert environment. The purpose of this race was to show that autonomous vehicles could execute a resupply mission through a priori unknown desert environment on a variety of road types with obstacles and limited interaction with other vehicles. When the race was first formed it was expected that it would be held several times before there would be a winner. The first race did not have a winner but the second race did. This showed that a robotic vehicle could navigate a desert or remote environment on a variety of road conditions, through a number of different kinds of obstacles, and through areas which had limited or no global positioning satellite (GPS) coverage. The third race, the Urban Challenge, raised the requirements significantly for the

competitors who had to develop autonomous vehicles which could traverse safely a 60-mile complex urban environment. Together the races showed that the technology exists for an autonomous supply mission to carry supplies between cities over a variety of road and environmental conditions. The following gives a brief overview of each of the races.

2.1 2004 DARPA Grand Challenge

The first Grand Challenge was conducted in March 2004 and was over an approximately 150-mile course in the desert between Barstow, CA and Primm, NV. This race was to simulate autonomous driving between cities in remote locations. The terrain could be desert, flat, or mountainous depending on the area. Typical obstacles when driving between cities may include bridges, underpasses, debris, paved roadways, poor roadways, and other vehicles (for the race other vehicles were suspended for passing).

There were 15 teams that qualified for the final event but there were no winners. The team that went the farthest was the Carnegie Mellon University Red Team's "Sandstorm" that went 7.4 miles. There was a number of reasons why vehicles did not finish. Some had initial problems at the gate and were withdrawn, some ran into obstacles, others ran off the course and were stopped. "Sandstorm" and some other vehicles were stopped when they got hung up on embankments along the course.

2.2 2005 DARPA Grand Challenge

The second DARPA Grand Challenge had a course that started and ended in Primm, Nevada. There were 195 applicants to the first DARPA Grand Challenge with teams from 36 states which included three high schools and 35 universities. From the entrants there were 43 semifinalists from 15 states and included one high school and 17 universities. The National qualification event (NQE) for the finals occurred at the California Speedway in Ontario, California on September 27–October 5, 2005.

The course included a number of obstacles including sharp hairpin turns, speed bumps, a tank track, rough roads, power poles, a tunnel, berms, and foliage. There was a wide range of vehicles that entered, including one two-wheeled motorcycle. Others included SUVs, specially built vehicles, a HMMWV, dune buggies, and an Oshkosh military vehicle. In this race five vehicles successfully completed the course with four of the five completing the race under the time limit. "Stanley", shown in Fig. 1, from Stanford University came in first and won the $2 million prize.

Fig. 1 The winning robot vehicle "Stanley" from Stanford University, CA in action (Image credit: DARPA)

2.3 2007 DARPA Urban Challenge

The Urban Challenge was the third in the series of robotic Grand Challenges conducted by DARPA. The first Grand Challenge simulated going between cities in a remote area and the goal of the Urban Challenge was to simulate a 60-mile supply mission through a busy city. Some of the challenges in driving through a city were to follow public traffic laws, safe entry into traffic flow, passing through busy intersections, passing stopped vehicles, U-turns, and finding an alternate route if encountering a blocked route. The race took place on November 3, 2007, in Victorville, CA on the former George Air Force Base. The prize awards were $2 million for the fastest qualifying vehicle, $1 million for the second fastest, and $500,000 for third place.

The goal of the Urban Challenge was to test the ability of robots to operate safely and effectively in populated, busy areas. It featured autonomous ground vehicles maneuvering in a mock city environment, simulating military supply missions, which had to negotiate the course along with approximately 50 human driven vehicles. The robotic vehicles had to complete a 60-mile course in less than six hours and had to obey California Traffic Laws while merging into moving traffic, navigating traffic circles, negotiating busy intersections, and avoiding stationary and dynamic obstacles.

The final event took place on November 3, 2007 at the urban military training facility located on the former George Air Force Base in Victorville, CA. The

location was rebuilt to provide a network of urban roads and thus, simulated the type of environment the military operates in when deployed overseas. The awards were for $2 million, $1 million, and $500,000 for the top three finishers that completed the final course within six hours. First place was CMU's "Boss", second place went to Stanford's "Junior", and third place went to Virginia Tech's "Odin".

3 Overview of the Urban Challenge

The purpose of the Urban Challenge was to develop a robotic vehicle that could traverse complex city traffic following public driving laws. The vehicles needed to traverse urban traffic with other autonomous and humanly operated vehicles at speeds up to 20 mph.

Vehicles had to demonstrate the capability to do a number of maneuvers, including merging into moving traffic, passing, parking, and following right of way rules at intersections and other points. In addition to these maneuvers, other vehicles had to be avoided, blocked roads detected, and routing between waypoints and blocked roads determined.

3.1 Track A and B Teams

There were two tracks competitors who could take in the Urban Challenge. Track A teams submitted a proposal through a BAA put out by DARPA and could request funding up to $1,000,000 which was paid as they passed milestones. Track B teams either did not request funding through the proposal process or did not win funding, but instead submitted an application to compete. Both Track A and Track B participants had to go through a process where they could be disqualified at each step.

For Track A Teams, all technologies that were developed under the received funding had to give the government use rights to it. There were 11 teams out of a total of 89 teams that received funding. The teams that either did not get funding or did not apply, could enter the race through an application process (Track B). These teams did not receive government funding and did not have to give up any rights to their technology. Track B participants first had to submit a technical paper describing their approach to developing the vehicle and the technology they would use. If the technical paper was accepted, at a later date these teams had to submit a five minute video showing the operation of their vehicle over a course executing specific maneuvers and operations. The rest of the process was the same for both Track A and Track B participants. Both participants that passed the previous milestones had a site visit by DARPA personnel, followed by the NQE comparable

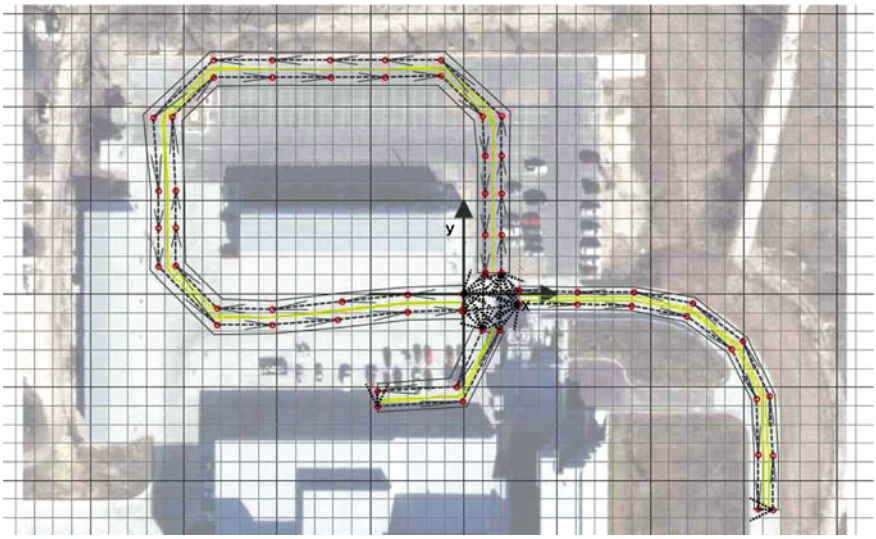


Fig. 2 Test area for the Team CarOLO site visit. Image credit: Team CarOLO

to the first and second Grand Challenges. Competitors who passed the NQE qualified for the final event.

3.2 Site Visits

Site visits from DARPA determined which teams would be invited to the NQE. Each team had to provide a course, based on DARPA specifications, on which the vehicles were to be demonstrated. Figure 2 shows the site visit course for Team CarOLO at the Southwest Research Institute in San Antonio, Texas.

The site visits tested the vehicles for basic safety capabilities, basic navigation, and basic traffic operations [6]. The basic navigation included being able to download a mission data file (see below and Appendix B) and start the autonomous mode within five minutes, follow the checkpoints as described in the data file, remain in the travel lanes during operation, not hesitate more than 10 secs during operation, maintain a minimum separation from obstacles, and being able to perform a U-turn in a limited spaced area.

The basic traffic operations included the safe passing of an obstacle vehicle blocking the own driving lane. In this maneuver the robot vehicle had to stop right before the blocking vehicle within a safe distance, use the direction indicator to change its lane for passing, and merge safely back into the own driving lane. Furthermore, the robot vehicle had to demonstrate correct behavior at a four-way intersection.

3.3 Safety

The overall safety of the competition, due to human drivers which had to interact with the autonomous vehicles and also the spectators, was the main and most important goal released by DARPA. To ensure this safety, the vehicles had to have several safety features. All vehicles had to have an "E-Stop" that would wirelessly either pause, stop, or run the vehicle [2]. For the NQE and final event the E-Stops were supplied by DARPA. The vehicles needed to be able to be paused in case an unsafe situation arose or if other vehicles became disabled or had to be removed. Vehicles also had to have a manual vehicle stop button easily reachable on each side of the vehicle. Vehicles also had to have audible alarms and warning lights that were activated to indicate their current operating state. In addition, the vehicles had to have operating brake lights, reverse lights, and turn signals, all of which had to function like a normal manned vehicle.

3.4 Data Files

The layout of the race course and the course that the vehicles were to take were given out as data files before the race [5]. These consisted of the route network definition file (RNDF) and the mission data files (MDF). Appendix A and B give the structure and example content of the files.

The RNDF contained the definition of the entire road course and open areas like parking lots where the vehicles could drive. It was given out to the participants 24 hours before the race. The file defined the valid road network by breaking the area up into road segments, waypoints, stop sign locations, lane widths, checkpoint locations, parking spot locations, and "free-travel zones" that are areas instead of roads. The zones were used to represent parking lots and areas that may contain moving or stationary vehicles or other obstacles. This file did not give the start and end points of the race. This information was provided by the MDF.

The MDF was given out five minutes before the start of the event and at least five minutes before the start of the second mission. The MDF file contained a sequence of checkpoints to be reached by the vehicle. Checkpoints were identifiable points within the RNDF. Furthermore, individual speed limits could be passed to each competitor for different parts of the RNDF.

3.5 Vehicle's Surroundings

All competing vehicles in the Urban Challenge had to complete a so-called mission which consisted of a sequence of given checkpoints. These checkpoints were arbitrary points from the RNDF. Using the RNDF likewise, the vehicles had to

interpret digitally provided lane markings according to the California traffic laws. At all times and whenever the traffic situation permitted the autonomous vehicle had to stay on the road and in its own lane. Whenever the vehicle detected a blocked road, it had to calculate an alternative route to reach the previously assigned destination. Besides regular paved roads, the vehicles could encounter unpaved or even rough areas within the road network.

Depending on the traffic situation, the autonomous vehicle had to choose a safe speed. Besides its autonomous operation in the stationary surroundings described by the digital map, the autonomous vehicle had to follow other cars moving in its own lane with a lower speed while keeping a minimum safe distance. Whenever it was allowed, the autonomous vehicles could pass a parked or stopped vehicle which was blocking its lane. However, passing was strictly prohibited at intersections.

When a vehicle approached an intersection it had to stop within one meter around the stop line. The stop line was listed in the RNDF and also clearly painted on the road. After passing an intersection or a T-junction, the autonomously driving vehicle had to be able to merge safely into already moving traffic. Furthermore, at T-junctions, the vehicles had to find a suitable gap between moving cars to pass through.

In addition to the road network, vehicles had to enter large areas which were called freely navigatable zones. Within these zones, no information about lanes or lane markings were provided. Moreover, stationary obstacles as well as moving vehicles could be encountered. The autonomous vehicle had to be able to find a parking spot which was previously assigned by the given MDF, park safely, and pull back out to continue the assigned mission.

The vehicles could use GPS signals, but there could be areas of GPS outage, so vehicles would need to be able to navigate safely without them for stretches. Moreover, the vehicles had to stay inside their lanes whenever the GPS signal had an insufficient quality.

Communications through wireless signals while the vehicle was operating was prohibitive. This was so that teams could not help their vehicles during the race by sending them instructions or updated data. This made sure that the vehicles were completely autonomous.

Out of scope for the autonomous vehicle was the detection of traffic signs and traffic lights and the detection of pedestrians and bicyclists. Moreover, the correct behavior on highways or in rough terrain was not a goal for the Urban Challenge [1, 6].

4 National Qualification Event

There were 35 teams that were invited to the eight day NQE [3]. The NQE, as well as the final event, were both located in Victorville, CA at the former George Air Force Base.

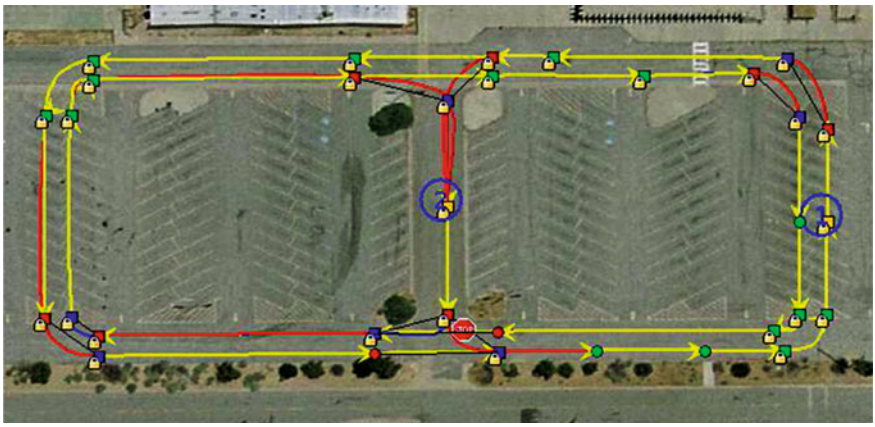


Fig. 3 Test Area A from the NQE (Aerial image credit: DARPA, augmented RNDF credit: Team CarOLO)

The NQE had three test areas (courses) [4]. The first area (NQE Test Area A) as shown in Fig. 3 was a test course that required the vehicles to merge into and out of two-way traffic in a circulating course. The course itself was shaped like a figure eight. Human drivers executed a choreography on the outer lanes of the eight providing repeatable gaps between them. The autonomous robots had to drive as many rounds as possible in the lower half of the eight finding suitable gaps to either merge into or pass through the moving traffic.

The second course (NQE Test Area B) depicted by Fig. 4 tested the vehicles' ability to stay within a lane on a 2.8 mile course while weaving in and out of obstacles. Furthermore, the autonomous vehicles had to find a previously assigned parking spot in a free zone between two cars and pull back out and continue.

The third course (NQE Test Area C) as shown in Fig. 5 evaluated the vehicles' abilities to negotiate traffic. There were two four-way intersections that included other vehicles at the stop signs, determine the order of vehicles to proceed based on traffic laws, recognize blocked roads, execute U-turns, and re-plan routes in case of blocked roads in real-time.

5 Final Event

Eleven teams qualified for the Urban Challenge Final (UCF) Event on November 3, 2007 which was also held on the former George Air Force Base in Victorville, CA. The final event was run over the entire previous three test areas of the NQE (Fig. 6), so a new RNDF file covering all three areas was given to the teams.

The following teams qualified for the UCF out of an initial field of 89 competitors:

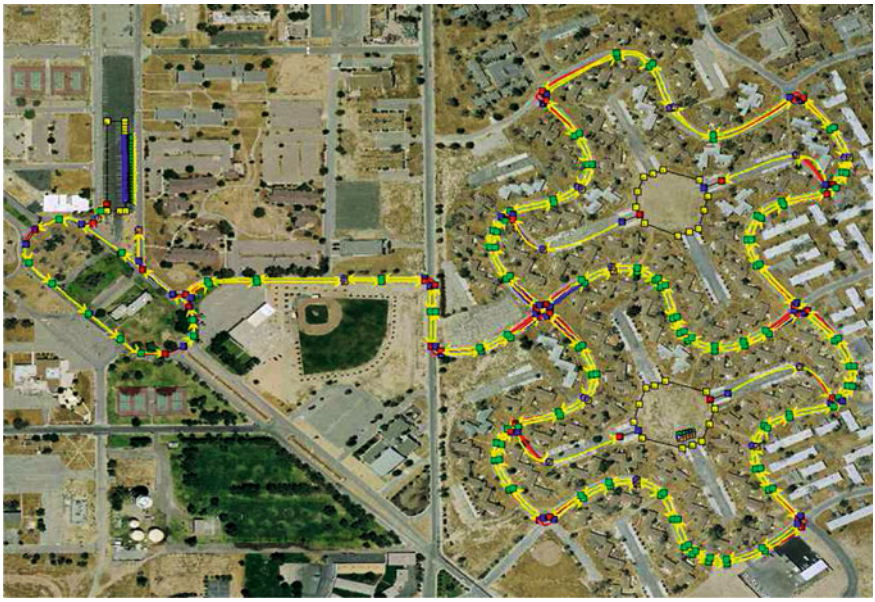


Fig. 4 Test Area B from the NQE (Aerial image credit: DARPA, augmented RNDF credit: Team CarOLO)

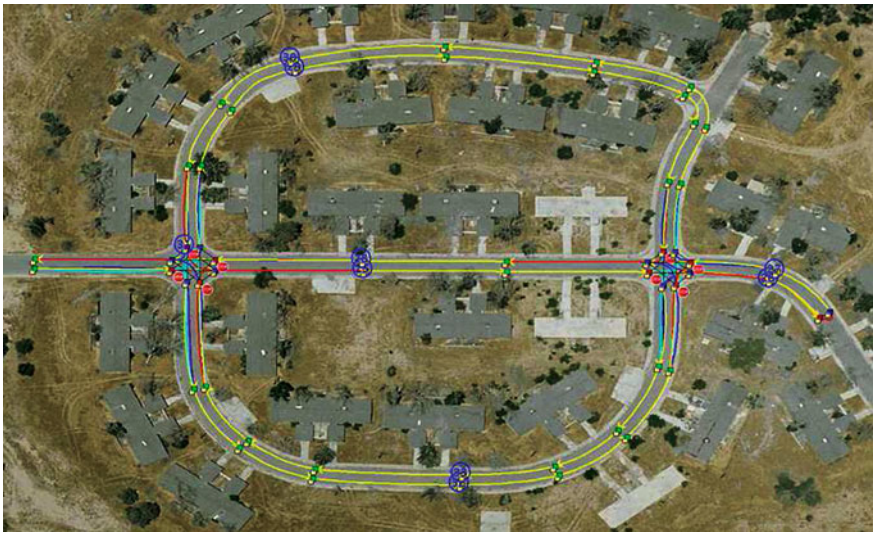


Fig. 5 Test Area C from the NQE (Aerial image credit: DARPA, augmented RNDF credit: Team CarOLO)

Team AnnieWay A spin-off of the Collaborative Research Center on Cognitive Automobiles from Universität Karlsruhe, Germany and the Universität der

Fig. 6 Course for the Urban Challenge Final Event (Aerial image credit: DARPA, augmented RNDF credit: Team CarOLO)

Bundeswehr München, Germany. Vehicle: 2006 Volkswagen Passat station wagon. Track B competitor. Result: Did not finish.

Ben Franklin Racing Team Autonomous vehicle developed by the University of Pennsylvania, Lehigh University, PA, and Lockheed Martin. Vehicle: 2006 Toyota Prius. Track B competitor. Result: Finished all missions but missed the 6-hour time limit.

Team CarOLO The entry of the Technische Universität Braunschweig, Germany. Vehicle: 2006 Volkswagen Passat station wagon. Track B competitor. Result: Did not finish.

Team Cornell Applicant of the Cornell University of Ithaca, NY. Vehicle: 2007 Chevy Tahoe. Track A competitor. Result: Finished all missions but missed the 6-hour time limit.

Honeywell/Intelligent Vehicle Systems Entry of Troy, MI and Dearborn, MI. Vehicle: Ford F-250. Track A competitor. Result: Did not finish.

MIT Team from MIT, Cambridge, MA. Vehicle: Land Rover LR3. Track A competitor. Result: Finished all missions and ranked fourth.

Team Oshkosh Truck The entry from Oshkosh, WI. Vehicle: Marine Medium Tactical Vehicle Replacement (MTVR). Track A competitor. Result: Did not finish.

Stanford Racing Team The entry from the 2005 winning team of the Stanford University, CA. Vehicle: 2006 Volkswagen Passat station wagon. Track A competitor. Result: Finished all missions and ranked second.

Fig. 7 The winning robot vehicle "Boss" from Carnegie Mellon University, PA in action (Image credit: DARPA)

Tartan Racing Applicant from the Carnegie Mellon University, PA. Vehicle: 2007 Chevy Tahoe. Track A competitor. Result: Finished all missions and ranked first.

Team UCF Entry from the University of Central Florida, FL. Vehicle: 1996 Subaru Outback Legacy 4WD. Track B competitor. Result: Did not finish.

Victor Tango Applicant from the Virginia Polytechnic Institute and State University, VA. Vehicle: 2005 Ford Hybrid Escape. Track A competitor. Result: Finished all missions and ranked third.

During the final event, each team had to complete three missions. The competitors were started sequentially with alternate missions. After completing one mission, the team was handed the MDF for the next mission. From these 11 finalists, six vehicles completed all three missions.

The winning entry for the 2007 DARPA Urban Challenge was the autonomous vehicle named "Boss" by Carnegie Mellon University, PA which is shown in Fig. 7. This vehicle had an average speed of 14 mph and finished the course after approximately 4 hour. The second place vehicle was 'Junior" from Stanford University and finished 20 minutes after "Boss". The third place vehicle, "Odin" by team Victor Tango, also came in 20 minutes after "Junior" finished the race.

6 Conclusion

Although the 2007 DARPA Urban Challenge raised significantly the requirements for all competitors to drive autonomously not only through a priori unknown environment, but also demanded them to interact with other moving vehicles.

Several independent approaches were demonstrated to tackle this task. Thus, the overall goal set by DARPA was fulfilled by at least four vehicles which were able to finish all three missions in the predefined time limit.

The following chapters are based on papers that originally appeared in two special sections of the AIAA Journal of Aerospace Computing, Information, and Communication [7, 8]. In the chapters, different aspects of the development of the autonomous vehicles that competed in the Urban Challenge are presented. Each contribution discusses one specific aspect ranging from sensor setup, data collection, pre-processing, and data fusion, up to data evaluation and interpretation to derive a driving decision for calculating the necessary set points for control algorithms. Furthermore, aspects for system and software development processes are presented and discussed to support the quality assured development of an autonomous system under a very tight schedule.

A number of approaches could have been used to complete the Urban Challenge, though different issues must be solved before this technology might be sold in consumer vehicles. First, sensors must be optimized to provide reliable and consistent information from the vehicle's surroundings at all times and situations. Second, a consistent theory for modeling and using human-like experience to support the deriving of driving decisions must be established. Third, legal issues in using autonomously driving vehicles in regular traffic must be discussed and solved to clarify questions of liability.

References

1. Defense Advanced Research Projects Agency: Urban Challenge Technical Evaluation Criteria (2006)
2. Defense Advanced Research Projects Agency: DARPA Urban Challenge E-Stop Guidelines (2007)
3. Defense Advanced Research Projects Agency: DARPA Urban Challenge Instructions for Semifinalists (2007)
4. Defense Advanced Research Projects Agency: Event-Guidelines (2007)
5. Defense Advanced Research Projects Agency: Urban Challenge Route Network Definition File (RNDF) and Mission Data File (MDF) Formats (2007)
6. Defense Advanced Research Projects Agency: Urban Challenge Rules (2007)
7. Rouff, C. (ed.): Second Special Section on the DARPA Urban Grand Challenge, vol. 4(12). American Institute of Aeronautics and Astronautics (2007)
8. Rouff, C. (ed.): Second Special Section on the DARPA Urban Grand Challenge, vol. 5(12). American Institute of Aeronautics and Astronautics (2008)

Part II
System and Software Architectures

Software Infrastructure for a Mobile Robot

Matthew McNaughton, Christopher R. Baker, Tugrul Galatali, Bryan Salesky, Christopher Urmson and Jason Ziglar

Abstract In this chapter we describe the software infrastructure developed by the Tartan Racing team to support the perception, planning, behavior generation, and other artificial intelligence components of Boss. The organizing principles of the infrastructure, as well as details of the operator interface, interprocess communications, data logging, system configuration, process management, and task framework are described with attention to the requirements that led to the design. The requirements were treated as valuable re-usable artifacts of the development process.

1 Introduction

A long-standing goal in the robotics community is to create software components that can be re-used by "plugging" them together to create software for new robotic platforms. Part of the motivation for this is the great expense of developing

M. McNaughton (✉) · C. R. Baker · T. Galatali · B. Salesky · C. Urmson · J. Ziglar
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
e-mail: mmcnaugh@ri.cmu.edu

C. R. Baker
e-mail: tallbaker@cmu.edu

T. Galatali
e-mail: tugrul@galatali.com

B. Salesky
e-mail: bsalesky@rec.ri.cmu.edu

C. Urmson
e-mail: curmson@ri.cmu.edu

J. Ziglar
e-mail: jpz@cmu.edu

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_2, © Springer-Verlag London Limited 2012

Fig. 1 Boss, the autonomous Chevy Tahoe that won the 2007 DARPA Urban Challenge

software, especially software for robots. However, it is difficult to realize savings from the opportunistic re-use of software components [2]. Components are not the only re-usable artifact of a development project. System requirements, architectural design, analyses, and test plans are a few of the many re-usable artifacts created during system development [2]. If all of the requirements of the software system could be anticipated in detail, it would be easy to construct it in one fell swoop of development. To the extent that the same requirements apply to robotics projects, they should be mined for re-use opportunities. Thus the detailed design of the system components of past projects, as opposed to the code itself, is also a useful resource. Fig. 1.

In this chapter we describe our best attempt at this goal; the software system of Tartan Racing's Boss [39] that won the DARPA Urban Challenge. Boss is built on a Chevy Tahoe chassis and incorporates a variety of sensors, including lasers, cameras and radars. Boss is capable of parking, passing stopped and moving vehicles, and safely interacting with traffic at intersections and at speeds up to 30 mph. Implementing these capabilities requires a relatively complex software system. Five main components make up this system: perception, mission planning, motion planning, behavioral reasoning ,and the software infrastructure.

The perception system detects and tracks moving vehicles, static obstacles and the road, and localizes the vehicle relative to a road map. The mission planning system computes the fastest route through an environment to a goal. The behavioral system makes tactical decisions about lane changes and interactions with traffic while executing the mission plan to reach a goal. This system was also

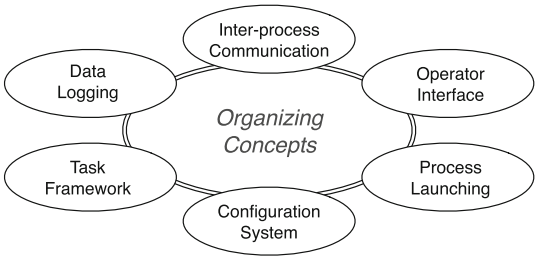


Fig. 2 Seven key pieces of the software infrastructure

responsible for the detection of, and recovery from anomalous situations, which was a key focus of our approach and critical to Boss's robustness. The motion planning system generates and executes feasible maneuvers given requests from the behavioral system and information from perception. Finally, the software infrastructure provides the necessary glue to allow each subsystem to communicate and to enable efficient testing and development through data logging, playback and configuration management.

During testing, the qualification events, and the Urban Challenge finale, Boss proved itself to be a capable vehicle. The team's intense test schedule included over 3,000 km of autonomous operation on test sites in three different states. During qualifications, Boss consistently performed well, completing each of the evaluation courses. The final event proved challenging, requiring Boss to invoke its error recovery systems on several occasions. Despite the difficulty, Boss was able to complete the course and do so faster than its closest competitor.

In this chapter we discuss the software infrastructure that enabled the rapid development, integration, and testing of this complex system. Our presentation of the Tartan Racing Urban Challenge System (TRUCS) software infrastructure is organized around seven key pieces (Fig. 2). These are the operator interface, the interprocess communications system, the process management system, the data logging system, the configuration system, and the task framework and data input and output libraries. The seventh piece is not a software component, but rather the organizing concepts of the system that guide the design and integration of the parts. We hope the reader will take away some lessons learned in the design of these pieces and the organizing concepts that were found to be effective.

TRUCS grew out of lessons learned through fielding the Red Team vehicles in the DARPA Grand Challenge races of 2004 [41] and 2005 [40], as well as other systems fielded in the past by the team members. The literature has few in-depth analyses of software infrastructure for mobile robot systems, aside from interprocess communications (IPC) packages [13, 14, 16, 33, 36]. The study of mobile robot software infrastructure is important because a well-crafted infrastructure has the potential to speed the entire robot development project by enabling the higher level architecture—the perception, planning, and behavioral reasoning systems. A software infrastructure done well lets the developers decouple their efforts from the overall development effort as much as possible. Designing the infrastructure with close attention to the requirements on the system can make the difference between success and failure.

The organization of this chapter is as follows. In Sect. 2 we survey related work in software infrastructure and re-use for mobile robot systems. We also look at work in infrastructure and re-use for other kinds of robot systems, as well as guidance from the field of software engineering on how to approach architectural design and re-use in software systems. Section 3 discusses the system software qualities and organizing concepts embodied in TRUCS, such as repeatability, flexibility, and statelessness. These qualities are not directly the property of any one module, but guided the development effort toward a principled overall design. Section 4 describes the requirements on and implementation of an operator interface, such as the developer's need for diagnostics. Section 5 discusses why TRUCS was a distributed system, and covers the design of the interprocess communication system devised to enable efficient communication between the dozens of tasks running on multiple computers in the robot. Section 6 covers the process launching and management system used to coordinate the startup, monitoring, and shutdown of the tasks responsible for carrying out a mission. Section 7 describes the data logging system for retaining and reviewing data generated during test runs. We discuss the organization of the software configuration system in Sect. 8. TRUCS required complex configuration data for the many sensors, tasks, missions, and hardware platforms the system ran on. Section 9 discusses the task framework and data input and output libraries that support the common functions of all tasks. We conclude with a discussion and considerations for future work in Sect. 10.

2 Related Work

In this section we survey efforts to describe, disseminate, and re-use robotic software systems. There are several published full-scale software frameworks for robotic systems [1, 13, 18, 20, 22, 23, 25, 27, 42]. Miro [42] is a task and communications library with device abstractions for wheeled robots. It has been used successfully by a research group unrelated to its creators to construct an outdoor mobile robot [8]. Player/Stage/Gazebo [13] offers a device abstraction interface, communications layer, and simulation framework for wheeled mobile robots. The CLARAty [25] framework from NASA is a framework for planetary rover software, including motion-planning algorithms for common rover platforms. CARMEN [23] is a task framework and component library for indoor wheeled robots and SLAM applications. ModUtils [15, 17] is a component architecture designed to isolate components from any knowledge of the components they are connected to and enable quick reconfiguration of components into different architectures for the perception, mission planning, and motion planning capabilities. The Microsoft Robotic Studio [22] is a contribution of software development tools into the hobbyist market.

There are a few evaluations of these software packages in the literature. Orebäck and Christensen [26] offer an in-depth analysis and evaluation of several architectures. They composed a set of reference requirements for a robot

application and implemented three software systems using different robot architectures on the same robot platform. The architectures compared were Saphira [20], TeamBots [1], and BERRA [21]. As a result of their experiments they formulated several design guidelines, for example that representations of sensor data and map data were an important but overlooked aspect of robot architecture; that modules should be individually addressable from the outside rather than mediating every command through a central deliberative component; that user interface, mission planner, and route planner should not be combined into a single component.

Broten et al. [8] document their experiences in selecting an architecture off-the-shelf for use in developing a non-trivial fielded mobile robot system. They compared Player [13], Miro [42], and CARMEN [23] in detail. The platforms were compared on the basis of documentation, available tutorials, support for multiple platforms, available sensor drivers, localization and mapping capabilities, and simulation capabilities. CARMEN was rejected for its lack of documentation at the time, though this improved by the time they published their paper. They found Player simple to use but lacking in multi-processing capabilities. Miro was selected for its powerful set of abilities. They found that the greatest barriers to re-use of robot frameworks was software complexity, poor documentation, and rigid system architecture.

Robotic software systems of significant size are almost always distributed systems. A few reasons are that the system may have multiple communicating robots, have CPU demands too high for one computer, or need to firewall between soft real-time and hard real-time requirements by putting them on separate computers. The literature on distributed systems outside of robotics is extensive. Waldo et al. [43] describe the essential differences between local and distributed computing. More specifically, robots are typically distributed real-time systems. Tsai et al. [38] describe the major challenges and some solutions for this type of system, though they give most of their attention to the development of custom computing hardware to support the demands of real-time response, which is out of scope for many mobile robotics applications, including TRUCS.

The interprocess communications (IPC) system is a core asset of a distributed robotic system, and there are many publications about them. The robotics literature offers some basic re-usable distributed communications components. These include IPC [36], IPT [14], CORBA [9], RT-CORBA [33], and NML (part of RCS) [12]. As with all components, architectural mismatches must be identified and mitigated.

Gowdy compared IPT [14], RTC [28], NML [12], NDDS (now RTI-Data Distribution Service [31]), MPI [24], and CORBA [9] for their suitability to robotics applications. He evaluated them in terms of portability, popularity, reconfigurability, ability to support different types of data flow models (e.g. query/response vs. broadcast), and ease of use. He found that IPT and RTC were most suitable for short-term projects with clearly-defined goals, and that CORBA might be suitable for long-term projects expected to evolve significantly in purpose. Further afield are software architectures for other types of distributed systems, such as Fielding's dissertation [11] on architectures for network software and the Representational State Transfer messaging paradigm.

Areas of robotics besides mobile robotics have distinct efforts to re-use software. Chetan Kapoor's dissertation [18] develops OSCAR, an architecture and component library for real-time control of advanced industrial robots. Orocos [27] is a research platform for component-based approaches to real-time robot control software. The related project Orca [5] is a component-based approach to re-usable robot software.

We focus in this chapter on the software infrastructure rather than the artificial intelligence software, but we should be aware of re-use efforts in this aspect. An example of these concerns is the way planning and perception functions should be divided into components and how communications between them should be structured. This perspective is important to the software infrastructure since it drives the facilities the software infrastructure should provide. A major architectural decision from the artificial intelligence perspective is whether to synthesize sensations into a coherent world model as CODGER did [37], or to hard-wire sensor readings directly into behavioral reactions as Brooks' subsumption architecture [7]. Another thrust of research effort is toward the ability to easily coordinate large-grained robot behaviors. Examples in the literature are the Task Description Language [35] and Colbert [19], used for specifying and ordering execution of robot behaviors. In terms of design lessons, Dvorak [10] claims that conventional notions of modularization in robotic systems creates difficulties in reasoning about resource usage.

We draw ideas from a rapidly maturing discipline for developing and analyzing software architectures in the mainstream software engineering research community [2, 34]. Software architecture rests on a discipline of articulating requirements, documenting the architecture, and consistently connecting the architecture to the requirements. A software architecture promotes or inhibits *quality attributes* such as "modifiability", "re-usability", and "availability", that describe properties of the system besides the actual functional results it computes. They are often not readily apparent from the structure of the code. *Quality attribute scenarios* [2] are a documentation format used to make these quality attribute requirements specific and testable. There exist typical approaches to achieving quality attributes. Bass et al. [2] term sets of such approaches *tactics*. They provide a checklist for the architect to ensure that all major approaches to a design problem have been considered and that their side effects have been anticipated and mitigated. In our research efforts we are moving toward greater uptake of these design ideas and disciplines to increase the quality of our software.

In the remainder of this chapter we describe seven key pieces of the software infrastructure for the Tartan Racing Urban Challenge System software infrastructure.

3 System Quality Attributes

In this section, we describe some of the major quality attributes that motivated the design of the TRUCS software infrastructure. A quality attribute is an adjective such as "modifiability", "re-usability", and "availability", that describes

properties of a software system that are not directly related to the functional results it computes [2]. Articulation and elaboration of the quality attributes is an important part of the design process. Quality attributes must be explicitly "designed-in". They can require collaboration throughout the system and attempts to hack them into the implementation after the architecture is fixed are usually expensive and unsatisfactory. We look at how the TRUCS infrastructure achieved, and in some cases did not achieve, some of these quality attributes.

3.1 Flexibility

The software infrastructure is designed to support the AI functionality such as perception and planning. A major requirement of the AI layer on the infrastructure is that it should be flexible in its ability to direct information flow. Capabilities desired of the robot can require that information be available for decision-making at unexpected places in the software structure. This can require radical redesigns of information flow and representation. ModUtils [17] is an architectural framework for infrastructure that is aimed specifically at providing this quality attribute, by allowing components to be configured at run-time to connect to their inputs and outputs using of shared memory, a TCP/IP socket, a file, or some other means.

TRUCS provides flexibility of module interconnections using a method similar to ModUtils. There is a code-level interface that the process uses to poll for new input from a stream of formatted messages. The messages retrieved through the interface could be coming from a file, the TRUCS IPC library, or any other method devised. Different message sources are implemented in individual shared object libraries which are dynamically loaded at run-time according to settings in the task configuration file.

3.2 Independence

The functionality of TRUCS was implemented by dozens of distinct tasks. The system should be robust to crashes in individual tasks. If a task crashes, the rest must continue to function. When the task is restarted, it must be able to smoothly resume its role in the system. This means that tasks must maintain *independence* from each other.

To achieve task independence, the architecture must eschew synchronous communication between tasks. A task cannot make a request to another task and wait for it to respond. If the recipient task of a request were unavailable, the requesting task would freeze waiting for a response. All tasks in the system could potentially freeze as a result. This indicated our choice of a publish/subscribe style to the interprocess communications system rather than a call-return style.

3.3 Statelessness

A task is *stateless* if its output depends only on recently received inputs. Statelessness helps to ensure that the function of a task in the system is not negatively affected if it is restarted.

If a task evolves an internal state over time that affects how it interacts with other tasks, a crash in that task could damage the integrity of communications between it and other tasks even after it restarts.

TRUCS had two tasks that were inherently stateful: a mission planner, which kept track of the current goal, and a road blockage detector, which tracked modifications to a prior road map due to road blocks. Their state was synchronized to disk to be recovered in case of a crash. Several other tasks accumulated state on a short-term basis, which was not retained in case of a crash. For example, when the robot arrives at an intersection, it must take its turn with other traffic. No effort was made to retain the relevant state variables in case of process crash. The restarted process assumed that the robot was the last of the waiting vehicles to arrive at the intersection.

3.4 Visibility

It is desirable for messages sent between tasks to be *visible*, that is, for it to be possible to understand the information being transmitted in a single message taken out of context. Visibility of communications allows the operator interface (discussed in Sect. 4) to be able to snoop on communications between tasks. If the tasks communicate in terse terms that depend on existing shared state between them, their messages may be impossible to interpret. Thus visibility and statelessness are related. Fielding [11] identifies scalability, simplicity, modifiability, visibility, portability, and reliability as key attributes of a network communications protocol.

Avoiding synchronous communications and incorporating statelessness mitigates the risk of partial failure, one of the four major architectural risks of distributed computing as opposed to local computing [43]. TRUCS avoided as much as possible having tasks maintain long-term state, and avoided patterns of communication that required tasks to query each other for that state. If necessary, state information was broadcast at regular intervals, whether requested or not. This was inefficient in terms of network bandwidth used when the information was not needed, but had the advantage that the communication was visible. Regularly transmitting all shared state, rather than only incremental changes, ensured that new observers could join at any point in time. A promising direction for future designs of the messaging paradigm should consider a message as an incremental transfer of state between tasks and provide additionally for explicit representations of the whole state.

3.5 Repeatability

Repeatability means that running the same program on the same input gives the same result. For typical single-threaded programs, repeatability is a necessary part of functional correctness. For a real-time distributed system such as TRUCS, repeatability is not necessary for correctness. Variations in IPC message arrival times or thread scheduling, due to non-determinism in the underlying hardware or operating system, can cause the system to behave differently, though still correctly. A simpler form of repeatability would allow the developer to debug an individual task by re-running it on logged data, presenting logged messages to a task in the same order in which they were received. The TRUCS infrastructure did not provide this capability, but we have identified it as a worthwhile addition to future systems. Without the ability to re-run tasks on logged input data exactly as it ran originally, developers must log excess diagnostic data in order to manually trace the execution of misbehaving processes.

4 Operator Interface

The operator interface used by TRUCS was called the Tartan Racing Operator Control System (TROCS). It was the face of the system, used for defining, selecting, and launching missions, viewing logged data, modifying the world during a simulation, and viewing the vehicle state while it was running. TROCS includes a 3D graphical view superimposed with geometric and textual diagnostic notations, and tabs containing text fields and 2D graphics displaying diagnostic information gleaned from throughout the system.

The need for developer diagnostics drives the design of the operator interface. A TRUCS developer might need to view any of dozens of pieces of data. These could be sensor data at any level of processing such as raw laser hit locations or identified moving obstacles; AI quantities such as predicted future positions of moving obstacles, or of hypothesized obstacles behind occlusions; or robot motion plans under consideration. The size and flux of this list argues for a framework that makes it easy for the TROCS developer to acquire data from the rest of the system and to create new views of data. It also suggests a need for the TROCS user to be able easily to select just a few views to display at once.

The following sections delve deeper into the requirements and implementation of TROCS. We found that the design of TROCS was sufficient for its purpose of providing online and offline diagnostics and simulation control. We would make two minor changes: factor out interaction tools from camera views, and create an internal event-bus mechanism to distribute data to drawables.

4.1 Viewing

TROCS displayed data in three ways. The first was a simple two-dimensional textual display of elementary quantities published by system tasks. Second was scrolling 2D graphs of time-varying numerical quantities such as the velocity controller's setpoint and response. These types of display would typically show data that did not need to be monitored in real-time, or that did not have a good spatial representation. Third was a real-time three-dimensional display of the world and the robot's position in it. Data was represented in this display with colored lines, points, and other geometric or textual symbols.

The TROCS display (Fig. 3) consists of a menu, a toolbar with buttons for loading, starting, and stopping mission scenarios, a 3D view panel showing the annotated world and vehicle, and a tabbed set of panes containing mostly textual diagnostic information. Some space is reserved for vital messages from the process launching system (described in Sect. 6). Users of TROCS interacted mainly with the three-dimensional view of the vehicle and road network. Developers created lightweight 3D "drawables" to display data whenever possible. A drawable produces a set of points, lines, and other graphical elements rendered with OpenGL, representing a specific piece of data. Drawables in TROCS were such things as boxes representing the predicted future positions of a moving obstacle, red squares representing the locations of lethal non-moving obstacles, rotating diamonds representing the lane that has precedence at an intersection, or multiple curves extending forward from the robot representing candidate paths to follow over the next 1 second. There were 23 different drawable classes providing 173 individually selectable data items for display. Little code was required to create a new drawable. During testing, the vehicle can switch between multiple modes of operation, for example from driving on a known road map to driving on an unknown dirt road that must be detected as it is driven on. The set of drawables of interest then also switches abruptly. This motivates the ability to save and switch between multiple configurations of drawables, though this was not implemented in TROCS.

In TROCS, a 3D view of the world is displayed through a camera. A camera is defined by a coordinate frame, which can be attached to the world or to the robot. The position of the camera can be moved within the coordinate frame by the user. If the camera is attached to the coordinate frame of an object that has a slow position update rate, the display may appear jerky and disorient the user. TROCS used an interpolated object position in these cases.

TROCS rendered the majority of three-dimensional diagnostic information by subscribing to messages produced by tasks and translating them into graphical representations. Another method used was for tasks to publish serialized OpenGL drawing calls directly on a designated channel shared by all tasks. The advantage of this approach was that it was easier for developers to write ad hoc visualizations while debugging functionality without adding a new message type and drawable that would eventually be obsolete. We identified several caveats against using this method beyond the early task debugging stage of development. Tasks using the

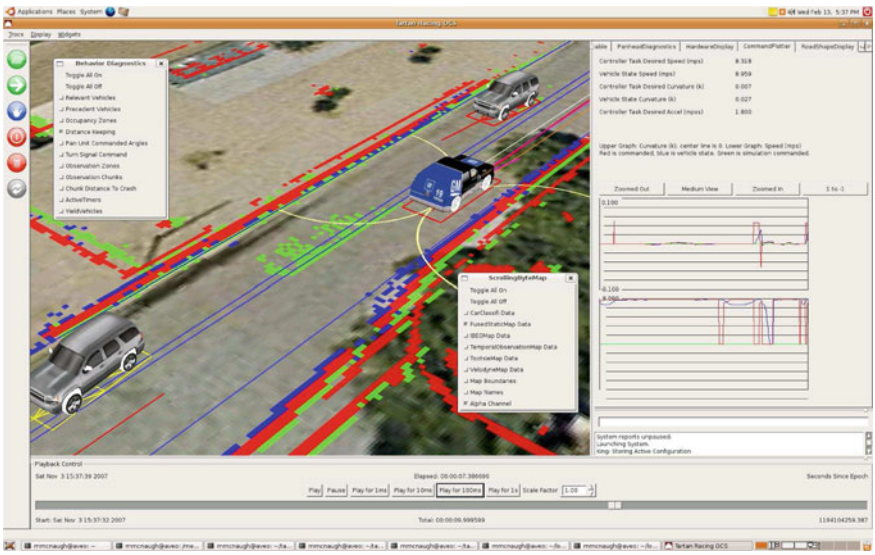


Fig. 3 The TROCS display

direct drawing method could disrupt TROCS in mysterious ways since any OpenGL command could be sent. It could be difficult for operators to distinguish between graphical elements produced by different tasks, since sources were not individually selectable as built-in TROCS drawables were. Extensive display code embedded in tasks could cause performance problems, slowing their operation to unacceptable levels. This also violated our guideline that robot control tasks should be unconcerned with display issues. Finally, diagnostic information represented directly as drawing commands could not be played back from logs with new and better renderings later.

4.2 Manipulation

The user should be able to interact with the 3D display by translating and rotating the view. Additional tools may be necessary to interact with the 3D display to get information about the world. For example, TROCS provided tools to measure the distances between locations in the world, and to display the lowest cost to traverse road elements.

In TROCS, user interaction in 3D space was provided by cameras. Interactions relevant to the road model were confined to the Road Model Camera, a world-fixed camera. Interactions relevant to the simulation, such as placing obstacles, were confined to the Simulation Camera, a robot-fixed camera. The Road Model Camera offered several interactions, which had to be disambiguated using

combinations of the Shift and Control keys and left- and right-clicks. Tools were always active when using a particular camera, so clicking to move the camera could accidentally invoke a tool, which could have undesired side effects on the world. The idea of factoring out a controller(tool) from a view(camera) is not new, but our experience confirms that future operator interfaces should incorporate this principle.

4.3 Movies

The operator interface is a natural place to make movies. Developers may need to discuss the system's behavior in a certain situation. Researchers need to promote and disseminate their work by giving academic lectures, send progress reports to sponsors, and solicit new grants. Videos support all of these efforts. The requirement to create movies of the operator interface has architectural consequences that should be taken seriously.

Third-party video screen-capture applications are available. However, they may be unreliable and unavailable or difficult to get working for less common platforms. They are often built on the assumption that the window to be captured has large areas of uniform color and changes infrequently. As a result, they may show unacceptable performance on large screen areas with detailed dynamic graphics. Playing back data at high rates through the operator interface to record a smooth-playing movie introduces timing problems that push the operator interface and the frame capture software to be integrated.

A movie has a characteristic frame rate and frame size. The frame rate is the number of frames per unit time displayed by the movie when played back at its intended speed. The replay rate is the number of real seconds represented by each second of the movie. The user of TROCS could control the replay rate of the data playback, and the frame rate of the recorded movie. However, movie recording happened in real-time and could not be modulated to compensate for hiccups in the data playback.

Slow disk accesses during playback caused TROCS to sometimes be choppy during replay. To remedy this in a future operator interface, four requirements follow. First, that the operator interface can directly control what time sequence of data is replayed. Second, that it knows what time is represented by the played back data. Third, that it can pause data playback until it is ready for more data. Fourth, that it knows when it has received all the data representing a particular range of time. This implies that it has to know which channels are actually playing back and that all the data has arrived from those channels. The testing and simulation facilities, which we do not describe here, benefit from similar enhancements. However, there is an architectural mismatch between these requirements and the anonymous publish/subscribe communications style used in the TRUCS IPC system (discussed in Sect. 5). Future work would have to consider these issues carefully.

TROCS used COTS (common off the shelf) video codec libraries to encode frames into MPEG movies. There are architectural risks when integrating COTS software into a system. We chose to link the libraries directly into TROCS. Instabilities in the libraries occasionally disrupted movie-making. Future designs should firewall the video encoding into a separate process.

4.4 Data Acquisition

TROCS did not have special status with the IPC system. It obtained data by the same mechanism as all other tasks. Drawables in TROCS subscribed to channels published by other parts of the system for the data they drew. Since TROCS functioned as one task within the framework of the IPC system, messages could only be consumed by one drawable. Most messages were only of interest to one drawable, which displayed every aspect of information it contained. For multiple drawables to subscribe to a channel, messages would have to be read by one drawable and communicated to other drawables in an ad hoc fashion, e.g. through a global variable. However, developers expected that multiple drawables could subscribe independently to the same messages, just as tasks could, so this was a source of bugs during development of some drawables. We note that this design broke the conceptual integrity of the publish/subscribe system. A future operator interface should have an internal anonymous publish/subscribe system to mirror the design of the IPC system.

5 Inter-Process Communications

Robotic software systems are frequently distributed systems, that is, the software runs as a set of separate processes, possibly on separate computers. Performance is a key motivation. Many mobile robot solutions require too much processing power or memory to run on a single CPU core, or even a single computer. Distributed systems can be more robust to program errors, and support parallel development efforts by multiple developers.

Many IPC systems have been devised for robotics, as discussed in Sect. 2. They are usually created for a specific project and its needs, and do not find wide-spread use. The design of the IPC system is often shaped by requirements directed at other parts of the overall system. Since important system requirements vary between robotics projects, there is reason to expect that IPC systems would not be re-used, even if they have well-factored implementations that are highly re-usable at the level of static code modules. Robotics IPC packages often use the anonymous publish/subscribe communication style [34] (also called implicit invocation or event bus) between processes rather than a call-return style, or remote procedure call. This communications style supports flexibility since tasks need not know explicitly about each other, so they can be added and removed from the system

Table 1 IPC Design Decisions

Design axis	Design choices
When communications channels may be created	SimpleComms supported creation of channels at any time, but the task configuration system required tasks to declare their channels at start-up
How participants are identified	SimpleComms required tasks to identify themselves with a unique string formed from host name and task name. This prevented the developer from accidentally starting two instances of the TRUCS processes on the same event bus
How participants discover the event bus	Tasks connected to a local SimpleComms server through a named unix pipe. SimpleComms servers on different machines discovered each other by broadcast on the local subnet
How data is marshalled from the process into messages, and unmarshalled when messages arrive	SimpleComms used Boost serialization [4]
How message queue overflows are dealt with	Tasks using SimpleComms had a fixed length queue. New messages that would overflow the queue were discarded
How messages are sent to multiple recipients	SimpleComms sent one copy of each message over the network to the SimpleComms server on each machine with a message subscriber
Whether messages are delivered reliably	SimpleComms used TCP to deliver all messages reliably

relatively easily. The Tartan Racing team developed its own IPC package, named SimpleComms, that was based on the anonymous publish/subscribe communications style.

The implicit invocation style in an interprocess communication context can be analyzed in terms of several design decisions. How those decisions are made depends upon the requirements of the overall system. In Sect. 3 of their book, Shaw and Garlan [34] discuss design decisions that must be made in an implicit invocation architecture. Table 1 summarizes the major design choices made for SimpleComms.

Figure 4 shows a sketch of the communication paths between processes in the TRUCS IPC system. Each machine has a SimpleComms server called `scs` that handles all communications between machines. Tasks participate in the IPC system by connecting to `scs` through a Unix domain socket. The `scs` processes find each other by broadcasting on the local network. The method of using a single `scs` process on each computer responsible for all network transmissions ensures that a message originating at one computer is transmitted to each other computer at most once, minimizing network bandwidth consumption. This enables scalability in the IPC system, so that it can operate unchanged as the number of processes and

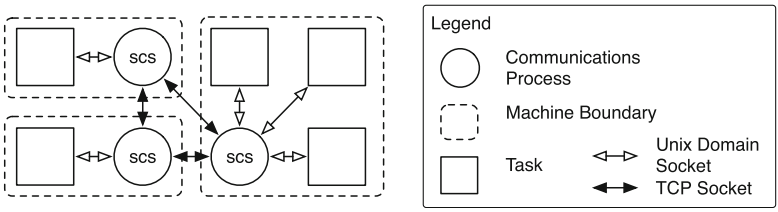


Fig. 4 The process view of the TRUCS IPC system

computers in the system increases. TRUCS transferred approximately 20 MB/s of data over the network. The volume would have been too much for a central hub that received and copied messages to all subscribers. Using broadcast or multicast to distribute data over the network could have reduced network load even more, but ensuring reliable delivery to all subscribers would have required more complex protocols.

The `scs` tasks on each computer copies messages to the local subscriber tasks. Data is marshalled into message structures explicitly, using the `Boost::serialization` package. An encoding should be quick to encode and decode and be concise. The `Boost::serialization` default binary encoding has these properties; we use it to encode messages for transport between computers. Since the Urban Challenge, we have found that the Protocol Buffers [29] library open-sourced by Google is also a good choice. Another option from the literature is Corba's [9] Interface Definition Language, which is used by the Miro framework [42].

A desirable quality attribute for a network protocol is *visibility*, which means it is easy to sniff the communications stream and display the data being sent over the network. For TRUCS it was convenient enough to display message contents in the operator interface.

The IPC system assumes that messages are guaranteed delivery and hides any packet loss in the TCP layer. When transport links are lossy, it could be beneficial to use a transport protocol such as UDP that does not insist on delivering all messages, if the message semantics for the channel are such that only the latest message is useful.

An important part of an IPC system is how the IPC libraries and its control flow interfaces with the task code. We discuss this aspect of the IPC system further in Sect. 9.

6 Process Launching System

TRUCS had dozens of distinct tasks running on ten computers. In such a large system it is necessary to automate the procedure of launching all of these tasks with the same configuration. In TRUCS, one process per machine, known as the

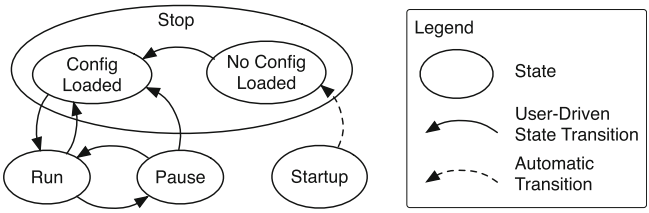


Fig. 5 State chart of the process launching system

`vassal`, is responsible for launching robot tasks for a specific mission. The `vassal` is started when the developer begins to use the robot and continues to run until he is finished. There is one `king` process that is responsible for mediating between the user at the operator interface and the vassals. It maintains a representation of the user's desired system run state, monitors the health of tasks, and distributes task configuration information to the vassals. It generates a name used to identify the run. This name is used to specify where logs generated during the mission should be stored. Data generated by the process launching system when no mission is running are not logged.

The process launching system must have access to the configuration files to determine a list of available missions, and learn which tasks are to be started. The `king` reads the configuration files, sends a list of available configurations back to the operator interface, and accepts the name of a mission selected by the user. The `king` extracts the configuration files needed for the given mission and sends them to the `vassal` tasks using `rsync` [30]. The `vassal` tasks run the mission tasks designated to run on their computers, and kill them when signaled by the operator. The `vassal` tasks are responsible for logging data related to program crashes, such as core files.

Figure 5 shows the states of the process launching system, maintained by the `king`. The Stop state has two sub-states, where there is either a configuration loaded or not. The transition from the Start-up state to the Stop/No Config Loaded state is automatic. Other transitions are driven by the user from the operator interface.

6.1 Process Health Monitoring

An unhealthy process typically either crashes or remains alive but does not respond to communications. The health monitoring system must be able to distinguish between these two states and be able to act in response to an unhealthy process. If the process has died, it may be relaunched according to some policy. If the process is in an infinite loop, it should be killed before being restarted. Processes may have different time allowances before being assumed to be stuck in an infinite loop. TRUCS processes reported on a `ProcessHealth` message

channel which the `king` listened to and summarized on a `HealthDisplay` channel, which was rendered by TROCS. A process which was not heard from for a set period of time was presumed to be stuck in an infinite loop. The `king` would send a command to the vassals to kill and restart these processes.

7 Logging System

Data generated by the system during a mission must be recorded for later review and analysis. It is a special challenge to handle the volume of data generated by large systems with many sensors and tasks. TRUCS had 84 publish/subscribe channels, of which 67 were logged at an aggregate rate of approximately 1 GB/min. A major impediment to logging large volumes of data is the rate at which hard drives can write the data. Space in Boss' equipment rack is at a premium, and hard drives in the hot moving vehicle wear quickly. Boss logs to two hard drives, each attached to a different computer designated for data logging. Once a test run is over, the data must be copied from the robot to long-term storage servers, and from there to developer laptops for them to run their analyses.

Given the limited amount of storage space available on a laptop and the limited speed at which the hard drive can read data, it is necessary to excerpt only the data necessary for a diagnostic task before it is copied from long-term storage to the developer's laptop. For most diagnostics tasks, only a subset of logged data channels are relevant. For example, if an oncoming car is not properly perceived, messages generated by planning tasks are not needed to diagnose the reason. Following the discipline of using stateless communications between tasks is an advantage here, since data generated long before the incident is not needed to interpret later messages.

TRUCS logging tasks were each responsible for logging one channel. Messages on a channel were logged to a single Berkeley DB [3] file, keyed on the time at which the logging task removed the message from its incoming SimpleComms messages queue. Note that these times were not exactly the same as the times at which other tasks processed the same messages, which introduced some uncertainty during debugging.

Logs are replayed by a dedicated task that reads messages from a given set of Berkeley DB log files. It publishes the messages back on the SimpleComms event bus at a controlled rate. Since publishers on the bus are anonymous, tasks that use the messages are unaffected by the fact that a different source is producing them. TROCS can be used to step forward through time and display the messages using the 2D and 3D drawables. This makes it easy to correlate events on different channels. New drawables can even be written to display new views on the same data, which could not be done if data was logged in the form of `printf` statements. Tasks besides TROCS can also use the data being replayed over the event bus. For example, logged sensor data can be used to test new perception algorithms.

8 Configuration System

TRUCS used a configuration system based on the Ruby [32] scripting language to manage the variety of task configurations that could run on Boss. Considerations in the design of this system are that there are many tasks, each with many tuning parameters. The tuning parameters may be correlated between tasks. They may also vary by environment or the mission undertaken in the environment. Selections of tasks to be run may differ across environments or missions, and many missions may be mostly the same but for a few small changes.

The organizing concept for the TRUCS configuration system is *inheritance*. Inheritance is used to generate variations on the configuration in two ways. First, "derived" configuration files can include "base" configuration files and then modify selected values. Second, files in a subdirectory corresponding to a mission replace files with the same name in parent directories. This allows the developer to generate a new mission that varies slightly from an existing mission by creating a subdirectory and filling it with configuration files that would replace the existing ones that sit at a higher level. Table 2 describes the configuration information that defines a mission.

A mission was represented as a leaf in a directory hierarchy containing Ruby script files. Figure 6 shows the typical levels at which configuration resources were kept. Each task read a Ruby configuration file named after the task. The Ruby file defined a tree of key-value pairs in the form of a nested hash table, which could then be read by the C++-based main task code.

9 Task Framework and Interface Libraries

Every task needs to read configuration data, manage initialization and shutdown, and set up IPC channels, among other things. These functions are done in the same way by all tasks, so a common library serves all tasks. This library provides a task framework, that is, a main program skeleton with specific places for a developer to write in the code unique to that task.

The TRUCS task framework runs the main task loop at a fixed rate set in the task configuration file. The task developer manually inserts a check in the task loop for a new message on each message interface the task subscribes to. The advantage of using a synchronous task loop is that it is simpler to code than an event-driven model where task code is run in response to message arrival. A significant disadvantage is that it can increase overall system latency, for example, between sensing an obstacle and reacting to it.

The message interface can read messages from the IPC system, directly from sensor hardware, from a file, or any other conceivable source. Each source is implemented in a shared object (`.so`) file which is loaded by the task library. An available source is selected in the task configuration file. Message sources can also be changed out at run-time.

Table 2 Information required to define a mission

Resource name	Content	Purpose
Route Network Definition File (RNDF)	A road map containing roads, their coordinates, number and extent of lanes, location of intersections, parking lots, stop lines, and checkpoints available for use in missions	To define where the robot is permitted to go and how it is to behave
Mission Definition File	A sequence of checkpoints in an RNDF	To define the sequence of locations that the robot must visit in order to complete its mission
Robot configuration information	The tasks to be run on the robot, such as the particular planners and sensor map fusers to use; which message channels to log; tuning parameters for individual tasks	To customize the behavior of the robot for the mission

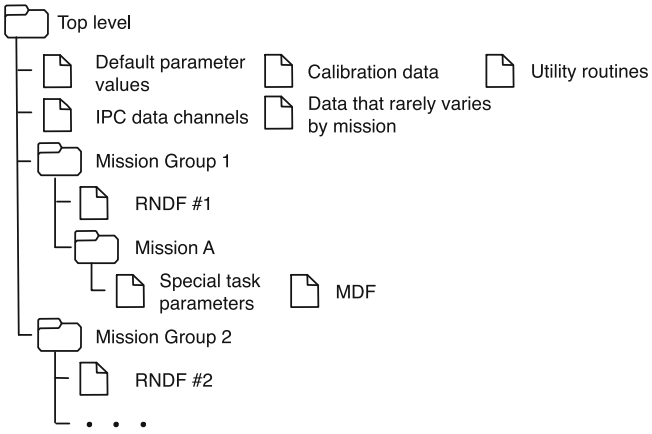


Fig. 6 A sample tree of configuration data

IPC system message sources create a set of threads to communicate with the SimpleComms server process on the local machine. Figure 7 shows the threads in the task library and how they interact with the input and output channels. The main thread reads from incoming message queues and writes result messages to the outgoing queue. Read/write threads push messages on and off the read and write queues through the local Unix pipe.

10 Conclusion

Fred Brooks contended [6] that "conceptual integrity is *the* most important consideration in system design" (italics original). As we build robots to perform more complex and dangerous tasks in uncontrolled environments, ensuring reliability

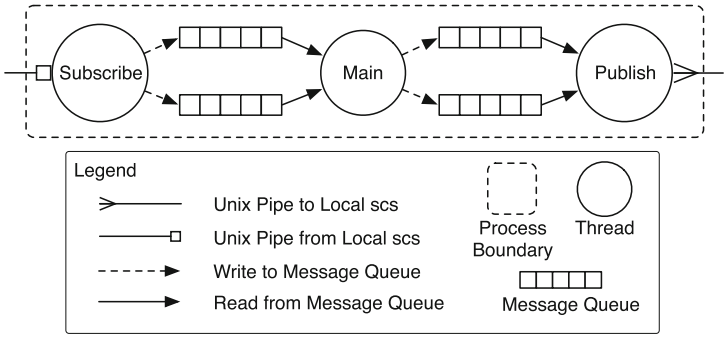


Fig. 7 Thread view of the IPC message library in a TRUCS task

becomes increasingly important. We believe that for a system to be considered reliable, it must be conceptually simple. Rigorous testing is vital, but testing alone cannot give us sufficient confidence in a robot's reliability. Since proving software correct is likely to remain impossible for the foreseeable future, we must rely on human-level reasoning about the robot's behavior, using a human understanding of its software. Conceptual integrity aids that human reasoning.

Discovering the right organizing concepts for the system and their best realization is an ongoing effort. We affirmed the event-bus architecture as a good fit for interprocess communications in mobile robot systems. We maintained system stability by adhering to the concept of independence among tasks, and visibility and statelessness in their communications. We also identified inheritance as the organizing concept for the TRUCS configuration system, though implemented in a broader sense than the language-based inheritance features found in C++ or Java. Exceptions and complications in the organizing concepts are inevitable. Attempts to graft on features that the architecture does not naturally provide for can cause architectural decay, that is, a loss of conceptual integrity, unless plans are made early in the design phase. As an example, the event-bus communications architecture is a good fit for mobile robot systems, allowing many independent tasks to communicate without needing to know the number or identities of their counterparts, or whether all messages sent have been received. However, for TROCS to render a movie frame, it needed to know that it had received all messages published before a given time, and for the process health monitor to tell whether all processes participating on the event bus were alive, it needed to know their identities. The designer should strive to become aware of architectural risks as early as possible in the design phase. This motivates the recording of system requirements as objects of re-use.

Looking forward, we see a need for a repository of re-usable requirements for mobile robot software infrastructure. The repository could contain candidate features to meet those requirements and their anticipated design implications, based on previous experience. Bass et al. [2] demonstrate this idea with *tactics*. We believe that an important measure of what has been learned from a software

project is the list of new requirements that were discovered during its development and that can be used for future systems.

New software tools are also needed to support new features. For example, an infrastructure that supports process-level repeatability, as discussed in Sect. 3.5, could greatly ease debugging and smooth the development of future systems, as well as potentially decreasing demands for storage space and network bandwidth. We must also be able to manage the evolution of state over long, complex missions in changing environments, all while retaining the benefits of stateless communications between tasks and the independence of tasks discussed in Sect. 3.3.

As robot systems increase in complexity, software infrastructure to support the development effort becomes increasingly important. We look forward to seeing additional work in this area.

Acknowledgments This work would not have been possible without the dedicated efforts of the Tartan Racing team and the generous support of our sponsors including General Motors, Caterpillar, and Continental. This work was further supported by DARPA under contract HR0011-06-C-0142.

References

1. Balch, T.: Teambots. Web site (2010). Retrieved 2 Feb 2010. http://www.teambots.org
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Boston (2007)
3. Berkeley, DB.: Web site (2010). Retrieved 2 Feb 2010. http://www.oracle.com/technology/products/berkeley-db/db
4. boost C++ libraries. Web site (2010). Retrieved 2 Feb 2010. http://www.boost.org
5. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Orebäck, A.: Software Engineering for Experimental Robotics chap Orca: a Component Model and Repository, pp. 231–251. Springer, Berlin/Heidelberg (2007)
6. Brooks, F.P. Jr.: The Mythical Man-Month Anniversary edn chap 4. Addison-Wesley, Boston (1995)
7. Brooks, R.: A robust layered control system for a mobile robot. IEEE J. Robot. Autom. **2**(1), 14–23 (1986)
8. Broten, G., Mackay, D. Barriers to adopting robotics solutions. In: Second International Workshop on Software Development and Integration in Robotics (2007)
9. CORBA middleware. Web site (2010). Retrieved 2 Feb 2010. http://www.corba.org
10. Dvorak, D.: Challenging encapsulation in the design of high-risk control systems. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (2002)
11. Fielding, RT.: Architectural styles and the design of network-based software architectures. Ph.D. Thesis, University of California, Irvine (2000)
12. Gazi, V., Moore, M.L., Passino, K.M., Shackleford, W.P., Proctor, F.M., Albus, J.S.: The RCS Handbook: Tools for Real Time Control Systems Software Development. Wiley, New Jersey (2001)
13. Gerkey, B., Vaughan, R.T., Howard, A.: The Player/Stage project: Tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003), pp. 317–323 (2003)
14. Gowdy, J.: IPT: An object oriented toolkit for interprocess communication. Techical Report CMU-RI-TR-96-07, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1996)

15. Gowdy, J.: Emergent architectures: a case study for outdoor mobile robots. Ph.D. Thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (2000)
16. Gowdy, J.: A qualitative comparison of interprocess communications toolkits for robotics. Technical Report CMU-RI-TR-00-16, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (2000)
17. Gowdy, J.: ModUtils. Web site. modutils.sourceforge.net (2010). Retrieved 2 Feb 2010
18. Kapoor, C.: A reusable operational software architecture for advanced robotics. Ph.D. Thesis, University of Texas, Austin (1996)
19. Konolige, K.: COLBERT: A language for reactive control in saphira. In: KI—Kunstliche Intelligenz, pp. 31–52 (1997)
20. Konolige, K., Myers, K.: The saphira architecture for autonomous mobile robots. Artificial intelligence and mobile robots: case studies of successful robot systems, pp. 211–242. MIT press, Cambridge (1998)
21. Lindström, M., Orebäck, A., Christensen, H.: Berra: A research architecture for service robots. In: International Conference on Robotics and Automation (2000)
22. Microsoft: Microsoft Robotics Studio. Web site. msdn.microsoft.com/robotics (2010). Retrieved 2 Feb 2010
23. Montemerlo, M., Roy, N., Thrun, S.: Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit. In: IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, pp. 2436–2441 (2003)
24. The message passing interface (mpi) standard. http: www-unix.mcs.anl.gov/mpi (2010). Retrieved 2 Feb 2010
25. Nesnas, I.A., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., Madison, R., Guineau, J., Shu, M.M.I.H., Apfelbaum, D.: Claraty: Challenges and steps towards reusable robotic software. Int. J. Adv. Robot. Syst. **3**(1), 23–30 (2006)
26. Orebäck, A., Christensen, H.I.: Evaluation of architectures for mobile robotics. Auton. Robots **14**(1), 33–49 (2003)
27. The Orocos Project. Web site (2010). Retrieved 2 Feb 2010. http://www.orocos.org
28. Pedersen, J.: Robust communications for high bandwidth real-time systems. Master's Thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1998)
29. Protocol Buffers. Web site (2010). Retrieved 3 Feb 2010. http://code.google.com/apis/protocolbuffers
30. rsync. Web site. samba.anu.edu.au/rsync (2010). Retrieved 2 Feb 2010
31. RTI Data Distribution System. Web site (2010). Retrieved 2 Feb 2010. http://www.rti.com/products/data_distribution/RTIDDS.html
32. The ruby language. Web site (2010). Retrieved 2 Feb 2010 http://www.ruby-lang.org
33. Schmidt, D.C., Kuhns, F.: An overview of the real-time corba specification. Computer **33**(6), 56–63 (2000)
34. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, New Jersey (1996)
35. Simmons, R., Apfelbaum, D.: A task description language for robot control. In: Proceedings Conference on Intelligent Robotics and Systems (1998)
36. Simmons, R., James, D.: Inter process communication. Web site (2010). Retrieved 2 Feb 2010. http://www.cs.cmu.edu/∼IPC
37. Thorpe, C., Hebert, M., Kanade, T., Shafer, S.: Vision and navigation for the carnegie-mellon navlab. IEEE Trans. Patt. Anal. Mach. Intell. 10(3):362–373 (1988)
38. Tsai, J.J.P., Bi, Y., Yang, S.J., Smith, R.A.W.: Distributed Real-Time Systems: Monitoring, Visualization, Debugging and Analysis. Wiley, New Jersey (1996)
39. Urmson, C., Anhalt, J., Bagnell, D., Baker, C.R., Bittner, R., Clark, M.N., Dolan, J.M., Duggins, D., Galatali, T., Geyer, C., Gittleman, M., Harbaugh, S., Hebert, M., Howard, T.M., Kolski, S., Kelly, A., Likhachev, M., McNaughton, M., Miller, N., Peterson, K., Pilnick, B., Rajkumar, R., Rybski, P.E., Salesky, B., Seo, Y.W., Singh, S., Snider, J., Stentz, A., Whittaker, W., Wolkowicki, Z., Ziglar, J., Bae, H., Brown, T., Demitrish, D., Litkouhi, B., Nickolaou, J., Sadekar, V., Zhang, W., Struble, J., Taylor, M., Darms, M., Ferguson, D.:

Autonomous driving in urban environments: boss and the urban challenge. J. Field Robot. **25**(8), 425–466 (2008)
40. Urmson, C., Anhalt, J., Bartz, D., Clark, M., Galatali, T., Gutierrez, A., Harbaugh, S., Johnston, J., Kato, H., Koon, P.L., Messner, W., Miller, N., Mosher, A., Peterson, K., Ragusa, C., Ray, D., Smith, B.K., Snider, J.M., Spiker, S., Struble, J.C., Ziglar, J., Whittaker, W.R.L.: A robust approach to high-speed navigation for unrehearsed desert terrain. J. Field Robot. **23**(8), 467–508 (2006)
41. Urmson, C., Anhalt, J., Clark, M., Galatali, T., Gonzalez, J.P., Gowdy, J., Gutierrez, A., Harbaugh, S., Johnson-Roberson, M., Kato, H., Koon, P.L., Peterson, K., Smith, B.K., Spiker, S., Tryzelaar, E., Whittaker, W.R.L.: High speed navigation of unrehearsed terrain: Red Team technology for Grand Challenge 2004, Tech Rep CMU-RI-TR-04-37, Robotics Institute. Carnegie Mellon University, Pittsburgh, PA (2004)
42. Utz, H., Sablatnög, S., Enderle, S., Kraetzschmar, G.: Miro—middleware for mobile robot applications. IEEE Trans. Robot. Autom. **18**(4), 493–497 (2002)
43. Waldo, J., Wyant, G., Wollrath, A., Kendall, SC.: A note on distributed computing. In: MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems—Towards the Programmable Internet, pp. 49–64. Springer, London, UK (1997)

The Sting Racing Team's Entry to the Urban Challenge

Matthew Powers, Dave Wooden, Magnus Egerstedt, Henrik Christensen and Tucker Balch

Abstract Autonomous navigation in urban environments inevitably leads to having to switch between various, sometimes conflicting control tasks. Sting racing, a collaboration between Georgia Tech and SAIC, has developed a modular control architecture for this purpose and this chapter describes the operation and definition of this architecture through so-called nested hybrid automata. We show how to map the requirements associated with the DARPA urban grand challenge onto these nested automata and illustrate their operation through a number of experimental results.

1 Introduction

Every mobile robotics project requires that several basic problems be addressed. The robot must sense the world and react to it. This requires reliable and effective computing, sensing, and locomotion hardware as well as robust software. On top

D. Wooden (✉)
Boston Dynamics, Waltham, MA 02451, USA
e-mail: dwooden@gmail.com

M. Powers · H. Christensen · T. Balch
College of Computing, Georgia Institute of Technology,
Atlanta, GA 30332, USA
e-mail: mpowers@cc.gatech.edu

H. Christensen
e-mail: hic@cc.gatech.edu

T. Balch
e-mail: tucker@cc.gatech.edu

M. Egerstedt
Electrical and Computer Engineering, Georgia Institute of Technology,
Atlanta, GA 30332, USA
e-mail: magnus.egerstedt@ece.gatech.edu

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_3, © Springer-Verlag London Limited 2012

Fig. 1 The sting racing retrofitted porsche cayenne entry to the DARPA urban challenge

of this, the urban challenge presented a new and unique problem in the space of the active robotics programs: the presence of multiple independent autonomous agents in a semi-structured but diverse and complex environment. We can contrast this with other contemporaneous DARPA projects, (e.g., UPI/Crusher, the LAGR project [2, 3]) in which the world is static and the robot itself is the primary moving object.

To model the dynamic complexities of the environment in the urban challenge, we choose to represent the various situations the robot can be placed in using a nested hybrid automaton, where each state corresponds to a particular scenario, defined at different levels of abstraction. This approach leads to an elegant decomposition of a very large problem into manageable subproblems. Above the automaton lies a mission planner, directing the robot to its next major objective. Below the automaton lies a control system composed of reliable parameterized control primitives responsible for the robot's actuation. Externally, perception processes provide fused sensing information based on a wide array of sensing modalities.

The sting racing team was composed of engineers from Georgia Tech, the Georgia Tech Research Institute, and Science Applications International Corporation (SAIC). Our base vehicle was a Porsche Cayenne (shown in Fig. 1) and modified to support a bay of COTS computers, LIDAR and RADAR scanners, cameras, GPS, and an IMU.

This chapter is broken into three main sections. In Sect. 2 the hardware components of our vehicle, including the base automobile, the sensors installed, and the computing resources are described. Section 3, describes how the world is sensed and how actions are planned, with special attention given to our approach

for hierarchically decomposing the various situations which the vehicle must be able to handle. Section 4, discusses testing and performance of the robot at several different sites including at the DARPA national qualification event in Victorville, California.

2 Hardware Architecture

Our approach to hardware design for the urban challenge was driven by a number of considerations. These considerations include the DARPA-defined design requirements, the requirements of available software approaches, and careful consideration of the demands of the task and environment. A discussion of the team's hardware approach and the factors driving the most important decisions follows.

2.1 Base Platform

Our base vehicle platform is a 2006 Porsche Cayenne, modified to be controlled via on-board computers. An AEVIT driving control system integrated by electronic mobility controls (EMC) provides primary servo control of steering and acceleration/braking as well as controls for secondary functions, including ignition, transmission, lights, and the parking brake. The drive-by-wire system allows for easy switching between autonomous and manual operation. Additionally, the hardware configuration allows four people to ride in the vehicle while in autonomous mode, which is a major convenience during evaluation and testing.

The computing system was comprised of eight Dual-Core Intel XEON 5120 processor-based computers, connected by dual gigabit ethernet networks. This computing power provides the resources to process the perception of the vehicle and the environment, control the vehicle, and log data. Power for the AEVIT servo and vehicle system controls is provided by the Cayenne 12 V DC system. All sensors and associated computer equipment are powered from an auxiliary, engine-driven, 24 V DC alternator configured to provide 75 A at idle. This exceeds the sensor suite and computer equipment power requirements of 50–55 A, 24 V DC during operation and provides additional power for future requirements. Batteries, contractors, circuit breakers, and the power distribution panel for the 24 V system are located in the rear vehicle compartment with the computer rack. DC–DC converters are installed to provide power for sensor and peripheral equipment requiring 12 or 5 VDC.

Roof-mounted equipment includes an amber safety strobe and emergency stop pushbuttons accessible from either side. The safety strobe and an audible warning signal provide an indication of autonomous operation. Antennas are installed for remote pause and disable receivers, and an 802.11 G WLAN interface is provided for developmental purposes. Other safety-related equipment includes internal

Fig. 2 **a** One of the two SICK LDLRS-1000 LIDAR scanners *blue* and 1 of the 7 SICK LMS-291 LIDAR scanners (beige). **b** The Riegl LMS-Q120 LIDAR scanner. **c** The four roof-mounted SICK LMS-291 units, mounted for use as in a "pushbroom" manner. **d** Rear-mounted SICK LMS-291

pause and emergency stop controls and the circuitry needed to disable the vehicle, power plant, sensor suite, and computer equipment when commanded.

2.2 Sensors

2.2.1 LIDAR

Our team installed four flavors of LIDAR scanners on the robot. The primary obstacle detection sensors on the system are the SICK LD-LRS-1000 planar scanners, two of which are mounted on either side of the front of the vehicle (see Fig. 2). These sensors have an angular resolution of 0.5°, effective range of about 100 m, and a spin frequency of 10 Hz. These two sensors cover nearly 360° field of view around the robot, with the exception of the region behind the robot's back bumper, which is covered by a different sensor.

In addition, three other planar LIDAR scanners are mounted on the front bumper. Two SICK LMS-291 scanners are mounted next to the LD-LRS-1000s and have a scan rate of 75 Hz, field of view of 180°, effective range of 40 m, and angular resolution of 1/4°. Mounted on the center of the front bumper is a Riegl planar scanner, with a field of view of 80°, angular resolution of 0.25°, and range of more than 120 m. These three scanners provide redundant coverage to the

Fig. 3 The eaton vorad RADAR units

scanning area of the LD-LRS-1000. Another SICK LMS-291 is mounted on the center of the back bumper, providing coverage of the rear of the robot.

Four SICK LMS-291 scanners are mounted on the roof in a "pushbroom" manner (e.g., see [12]). These provide detection of geometric features of the road like curbs, and are also used to detect vehicles in front of the robot. We also attempted to use these to detect road markings based on reflectivity information, but the signal to noise ratio was not sufficient for this to be a reliable source of information (Fig. 3).

2.2.2 RADAR

For collision warning radar, we selected an Eaton Vorad EVT-300 as a means of providing reliable detection for oncoming vehicles at large distances. The EVT-300 emits 3 mW of RF power at 24.725 GHz, with 1 MHz bandwidth. Able to detect and track as many as 20 objects at up to 150 m, it can update seven tracks at 15 Hz. A motorcycle-sized target can be detected at distances as great as 110 m. Target range ($\pm$3 ft), velocity, and azimuth are included for each tracked target. With approximately a 12° field of view, multiple EVT-300 units are required to provide adequate warning for the front and sides of the vehicle. One front-mounted unit provides redundancy for the Riegl, as well as the complementary capability to maintain tracks independently of other system software. Four side-mounted units are placed near the opposite ends of the front bumper to provide indication of approaching traffic as the vehicle pulls into an intersection or out of a side road. Figure 4 shows the sensor configuration mounted on the vehicle and Fig. 5 shows the sensors.

2.2.3 Cameras

Six Prosilica GC-650 Gigabit ethernet color cameras provided images to the front, sides and rear of the vehicle. They are capable of providing 12 bits of intensity

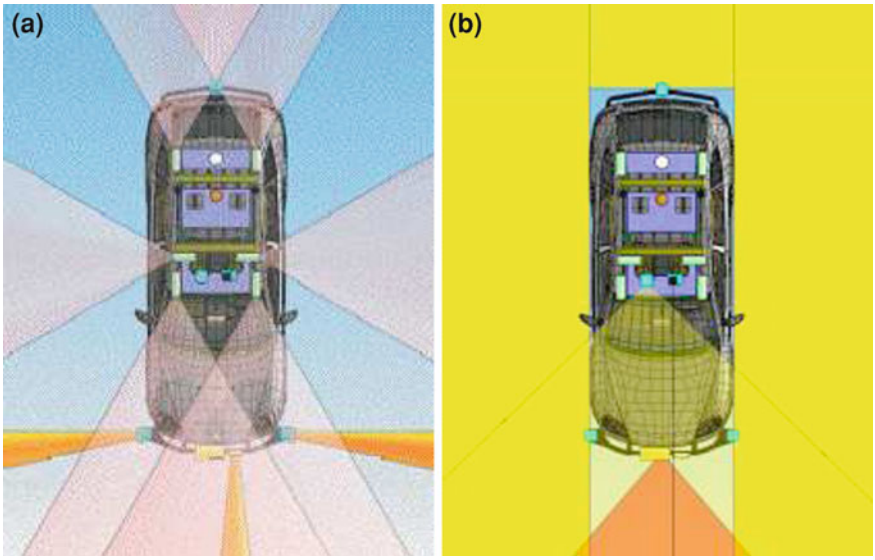


Fig. 4 In **a**, sensor placement and coverage for six Procsillica GC 650 color Gigabit ethernet cameras *pink* and three EVT-300 radars (*orange*). In **b**, sensor placement and coverage for five SICK LMS 291 LIDARs (*yellow*) and one forward-facing Riegl LMS-Q120 LIDAR (*orange*)

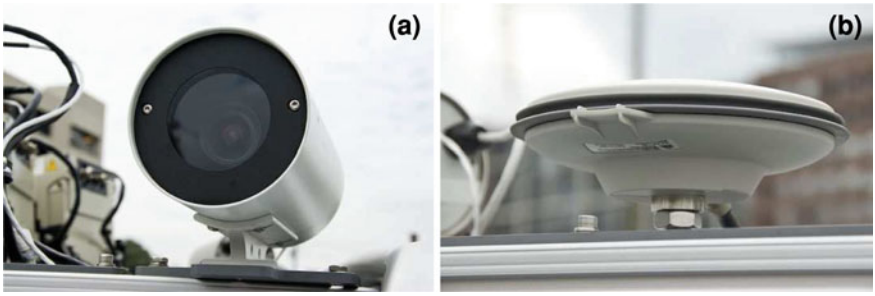


Fig. 5 **a** One of the six roof-mounted prosilica cameras. The forward-mounted cameras were used as the primary lane-sensing tool. **b** The GPS antenna, mounted on the roof of the vehicle

dynamic range per pixel, and the fast Ethernet interface supports high frame rates. All six cameras were roof-mounted in environmentally sealed enclosures. The two forward-looking cameras are used for road following.

2.2.4 GPS and Inertial Sensors

The vehicle was equipped with the NovAtel SPAN inertial navigation system, consisting of the NovAtel Propak G2plus GPS receiver with roof-mounted antenna and a Honeywell HG1700 AG17 inertial measurement unit mounted over the rear

Table 1 IMU specifications for SPAN INS

Gyro input range	$\pm 1{,}000°\ s^{-1}$
Gyro ratebias	$10.0°\ h^{-1}$
Gyro rage scale factor	150 ppm
Accelerometer range	$\pm$ 50 g
Accelerometer linearity	500 ppm
Accelerometer scale factor	300 ppm
Accelerometer bias	3.0 mg

Fig. 6 The rack of eight computers mounted in the trunk of the cayenne

differential, inside the vehicle. The GPS unit receives both L1 and L2 signals and is capable of pseudorange differential corrections. OmniSTAR High Precision corrections are used to improve GPS accuracy. The IMU allows the vehicle's position to be estimated during periods of GPS dropout, as are expected in an environment with limited view of the sky. The HG1700 IMU specifications are given in Table 1. The SPAN technology described by NovAtel [7] combine WAAS-corrected GPS readings with IMU data to provide position accuracy with 0.8 m CEP, velocity accuracy of 0.02 m/s RMS (nominal), attitude accuracy of 0.015° (pitch or roll) and 0.05° (yaw), and acceleration accuracy of 0.03 m/s^2. In referenced tests, an error of 1–3 m was seen during a 60 s dropout period, depending on how many GPS satellites were blocked. Additionally, the time to reacquire a GPS position after a dropout was improved dramatically, from 11 to 1 s, by keeping an estimated position based on IMU data.

2.3 Computing Resources

A rack of eight computers was mounted in the trunk (Fig. 6). Each computer was equipped with a dual-core Intel XEON 5120 processor and 4 GB of RAM. A Gigabit Ethernet network provided connectivity between all the computers, in addition to providing an interface to the six roof-mounted cameras and the four bumper-mounted RADAR units. The eight computers were utilized in the following way:

- Reactive steering and speed control (one computer)
- Mission, route, and situational planning (one computer)
- LIDAR/RADAR processing (two computers)
- Static and dynamic obstacle detection and mapping (one computer)
- Lane tracking (one computer)
- Data logging (one computer)
- Spare (one computer)

Communication between processes on one computer was implemented using shared memory, while communication between processes on different computers was implemented over the ethernet network. Highlights of functionality of these software processes are discussed in detail in Sect. 3.

3 Technical Approach

Our overall approach was motivated by the ambition to drive on road networks with sparse GPS waypoints (one marker per 20 m) with poor GPS reception. This necessitated driving based on local perception (LIDAR and camera based sensing) rather than driving based on detailed prior knowledge of the road or precise global positioning of the vehicle. Generally speaking, our aim was to produce a system that could find its own way while driving down road segments, negotiate intersections as it came to them, and deal with unexpected events along the way.

A final concern central to our design was the notion that the robot must strictly obey the rules regarding vehicle spacing and separation as specified by DARPA. This meant that the robot would not drive next to another vehicle or any other obstruction that brought it closer than 1 m on the sides or several meters in front (depending on vehicle speed).

3.1 High-Level Architecture

From a high-level standpoint, the software architecture follows a hierarchical, hybrid approach. Notable is the combination of a sequential, deliberative and reactive layers, combined in a manner similar to classical three-tiered systems [1, 5], as shown in Fig. 7. The mission-level strategic planner provides a symbolic plan over the high-level graph-based map, recommending a mode of operation (such as drive along paved road, drive along unpaved road, drive through intersection, or drive in unstructured area) along each edge in the graph. This plan is passed on to the tactical deliberative layer and the sequential layer in the form of a commanded task.

The tactical deliberative layer maps the commanded task to a spatial planning goal. For some tasks (e.g. drive to a point in an unstructured environment), the

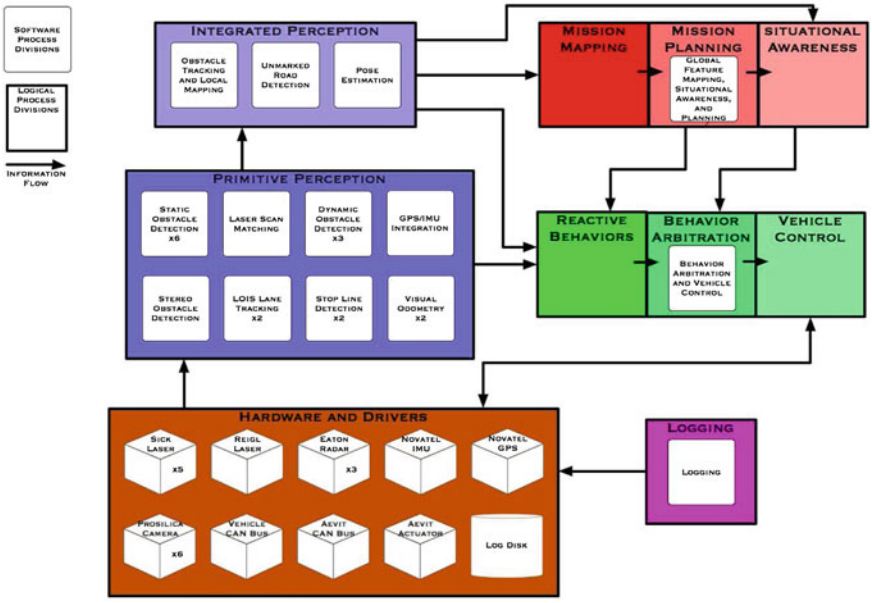


Fig. 7 Software processes used within the Sting software architecture and their relationship to the conceptual architecture presented in Fig. 1. *Smaller boxes* represent divisions of labor between software processes (e.g., static obstacle detection). *Larger boxes* represent divisions of labor within the conceptual architecture (e.g., primitive perception)

goal is a location in the global map. For other tasks (e.g. follow paved road), the goal is to navigate along a perception-based signal (e.g. the detected road) as far as the horizon of usable perception.

The sequential layer builds upon the paradigm of the finite state automata, implementing a nested hybrid automaton [13]. Nested hybrid automata extend the functionality of hybrid automata, and are explained in Sect. 3.6. Input from the strategic deliberative layer is interpreted as discrete input to the transitions in the top level automata.

The reactive layer is implemented by a voting scheme over circular arc paths at increments of curvature from the robot's current position. The state of the sequential layer is mapped to a weighting scheme over the voting behaviors, and the plan produced by the tactical deliberative layer is interpreted as a series of goal locations by one or more of the behaviors.

3.2 Static and Dynamic Obstacle Detection

The robot is equipped with three types of obstacle sensors: horizontal plane LIDAR scanners, pushbroom LIDAR scanners, and RADAR scanners. Each sensing

modality is processed separately to detect relevant features. The features are then fused to create a coherent representation of the robot's surroundings.

A point cloud is collected from the horizontal LIDAR scanners at regular intervals. This cloud is segmented into a list of distinct objects based on spatial spacing. The objects in each such list is tracked from iteration to iteration based on a greedy matching algorithm. A first-order heuristic-based estimator computes a world-frame velocity for each object, based on its position change over time.

The downward facing pushbroom LIDAR data passes through a filter to extract relevant features such as car profiles and curbs from the ground data. These features are incorporated with the horizontal plane LIDAR data at the initial object detection level. The Eaton Vorad RADAR devices return position and velocity estimates for up to 20 independently tracked objects, per device. This is combined with tracked object information to supplement the object list and improve the velocity estimate generated by the LIDAR data.

As a final step, each object is identified to be static or dynamic, based on its velocity estimate over its lifetime. In addition, in order to support various components of the planning and control modules, we use the geometric and velocity information of each object to assign a classification. The classification comes from the set *car*,*wall*, *ground*, other. For example, the *car* class of objects are used for establishing precedence at an intersection.

3.3 Lane and Road Detection

The primary behavior of the robot is to drive down a road. In order to do so, the robot must know its location relative to the road. GPS positioning is not accurate enough to perform this function, even with OmniSTAR High Precision corrections. Furthermore, the GPS lane information and satellite imagery supplied by DARPA could not necessarily be relied upon to be as precise as needed.

We pursued an approach that estimated the relative position and curvature of the lane based on visual clues from monovision cameras [4, 8, 9]. This approach is capable of detecting not only the white and double yellow lane markings, but also could detect the apparent luminance deviation that occurs on a curb, as demonstrated in Fig. 8. This approach has the benefit of being completely independent of any external information source (such as a route network definition file (RNDF) or a priori satellite imagery).

Because the vision system could be confused or washed out, often without cognizant failure, we used a statistical combination of the GPS lane information with the visual lane perception to form a fused lane estimate. Essentially, when the visually perceived lane was within a corridor surrounding the GPS lane estimate, the vision-based lane was given greater weight. When the lane estimate deviated, more weight was given to the lane estimate based on GPS and clues from detected curbs. In addition, the speed setpoint of the robot was reduced.

Fig. 8 Two examples of the paved-road lane tracking used on the sting vehicle. The detected lane is indicated by *red dots*

3.4 Intersection and Vehicle Detection

One of the key perception challenges of the urban challenge is to determine when cars are present or approaching an intersection. This requires that the robot be able to detect stationary and moving cars, estimate their position relative to the intersection, and properly ignore other detected environmental features and clutter.

The robot knows that it is approaching an intersection based on its distance to specially-marked node in the RNDF graph (see Sect. 5). As it comes to within 15 m of the stop line, it enables the vision-based stop line detector for precise alignment with the intersection, since the GPS information is not reliable or accurate enough for this Fig. 9.

To determine intersection precedence at a four-way stop intersection, we first formed the position and orientation of simple polygon bounding boxes at the known given position of stop lines as indicated by the RNDF. Then, the estimated positions of vehicles are pulled from the dynamic obstacle map, as described by Sect. 2. When the robot stops at a stop line, it uses the presence of these vehicles in the intersection-aligned bounding boxes to establish its own precedence at the intersection. To determine when it is its turn to proceed through the intersection, the robot detects the movement of cars through the intersection using the estimated position of *car* obstacles and a polygon representing the intersection's interior.

Intersections where the robot is required to come to a stop but cross traffic is not (i.e. a two-way stop) is particularly challenging. Because the on-coming traffic can be approaching at 30 mph, and the robot needs a 5 s gap (at least) to merge into traffic, the robot must be able to detect and distinguish oncoming cars at 67 m. Adding to the problem, the robot must identify which approaching cars are truly coming to the intersection and which are simply on a neighboring, disjoint road segment. For those cars approaching the intersection, their time of arrival must be calculated. We do this by localizing the detected vehicles to the RNDF, and by

Fig. 9 A snapshot of the robot (shown distant) waiting on a human-operated car to exit a four-way stop intersection. This occurred at the day-to-day test site (see Sect. 4)

estimating their forward velocity, estimate their time of arrival. So, for two-way stop intersections, the robot uses the position of vehicles stopped at the intersection at stop lines for precedence, while evaluating the time of arrival of the oncoming cars to which the robot must yield.

We use similar logic and the same perceptual elements to evaluate when the robot can change lanes on a multi-lane road. This lets the robot assess when it is safe to move between lanes that are moving in the same direction without cutting off another vehicle on the same road segment.

3.5 Route Planning

As stated in Sect. 1, the overall decision making process of the robot is decomposed into three subcomponents. A route planner assesses where the robot is now, and given the next goal location (from the DARPA mission data file), plans a path over the RNDF to get to that goal. The RNDF is a simple weighted graph structure over which an implementation of Dijkstra's algorithm is sufficiently fast for mission-level path planning.

The RNDF is also used to encode temporary changes to the road network. For example, when a section of road is found to be blocked (as determined by lower-level control system), this information is lifted back up to the RNDF planner where a high cost is assigned to the edge in graph associated with the section of blocked road. This high cost is large enough to induce the robot to drive anywhere else over the RNDF to get to the goal, but if no other passage is found (or all other passages are found to also be blocked), the robot will try again to re-traverse a section previously found to be blocked.

3.6 Situational Awareness: Nested Hybrid Automata

A major difference between the DARPA urban challenge and the DARPA grand challenge was the additional complexity of the task in the urban environment. In addition to navigating through a static environment, the urban challenge required autonomous vehicles to interact with other moving vehicles (autonomous or otherwise), follow visually-marked lanes, and obey traffic laws. A major challenge in designing the software architecture for the urban challenge was providing the expressiveness necessary to capture the complexity of the task. Our approach to this challenge was to describe the task in a model based on a modified hybrid automaton.

A hybrid automaton is a model that captures both the continuous and the discrete aspects of a dynamical system. In particular, a continuous state (typically the position and velocities of the car) evolves concurrently with a discrete state (the current mode of operation), and we follow the standard definition of a hybrid automaton [6] as a tuple $H = (Q, X, E, U, f, G, R, x_0, q_0)$, where

- Q: the set of discrete states
- X: the continuous state space
- E: the set of events that can trigger transitions between different discrete states
- U: the input space
- $f : Q \times X \times U \rightarrow TX$: encodes the evolution of the continuous state x as $\dot{x} = f(q, x, u)$
- $G : Q \times Q \times X \times (E \cup \epsilon) \rightarrow \{0, 1\}$: gives the guard conditions that triggers transitions between discrete states. In particular, a transition occurs between q to q' if the continuous state is x, the external event is $e \in E$ or possible the "empty event" ϵ (no event happened) if $G(q, q', x, e) = 1$
- $R : Q \times Q \times X \times E \rightarrow X$: encodes the reset condition, in that the continuous state is reset to $R(q, q', x, e)$ when the system transitions from q to q' at continuous state x under event e
- $q_0 \in Q$:initial discrete state
- $x_0 \in X$: initial continuous state

Different definitions of such hybrid dynamics have been given, but they all share these basic building blocks in terms of continuous and discrete dynamics, guards, and resets.

An example of this is seen in Fig. 10. In the figure, the discrete state starts out at q_0 and the continuous state evolves from x_0 according to $\dot{x} = f(q_0, x, u)$ until time τ. At that time, the continuous state is at $x(\tau_-)$ and event e happens. The guard condition $G(q_0, q', x(\tau_-), e)$ becomes one and the discrete state transitions from q_0 to q'. The continuous state is reset to $x(\tau_+) = R(q_0, q', x(\tau_-), e)$, from which it evolves as $\dot{x} = f(q', x, u)$.

Certainly, the task of completing the urban challenge could be expressed as a single, very large hybrid automaton. One would start by enumerating all the necessary control modes for the vehicle, and then define the connectivity between

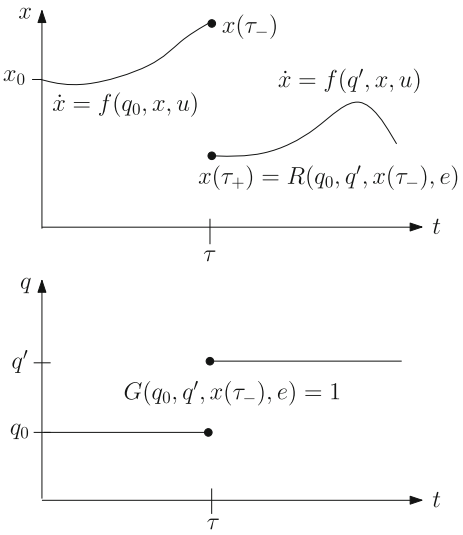


Fig. 10 Evolution of a hybrid automaton

the control modes based on perceptual signals. However, for a sufficiently complex task, this representation quickly becomes cumbersome to work with.

While the distributed structure of the hybrid automata keeps execution by the control system tractable (since the control system needs only execute the current control mode and check the current mode's out-going guards, which increase linearly with the number of modes), management of the system by developers can quickly become difficult. Our solution to this issue is the development of a nested hybrid automaton, which makes use of the natural structure of the task and environment to decrease the complexity by grouping related control modes into a hierarchical structure.

A nested hybrid automaton differs from a hybrid automaton in that it is defined recursively. Just as in a hybrid automaton, a finite automaton maintains the discrete state of the system. These discrete states can be interpreted as modes of operation for the robot, mapping to a function relating continuous state to continuous control. However, in a nested hybrid automaton, each state in the finite automaton represents either a discrete state (as in the hybrid automaton) or another sub-automata which maintains a more fine-grained representation of the discrete state in a hierarchical manner. The advantages over a standard hybrid automaton is an increased compactness of representation, modularity and isolation, as well as the ability to express discrete state transitions at different levels of abstraction asynchronously.

As an example, take the highest-level states used to define the operation of the Sting vehicle:

- `Operator-run`: navigate through the urban environment.
- `Operator-pause`: stop the vehicle and wait for operator input.
- `Operator-stand-by`: sound the warning siren before transitioning into `operator-run`.

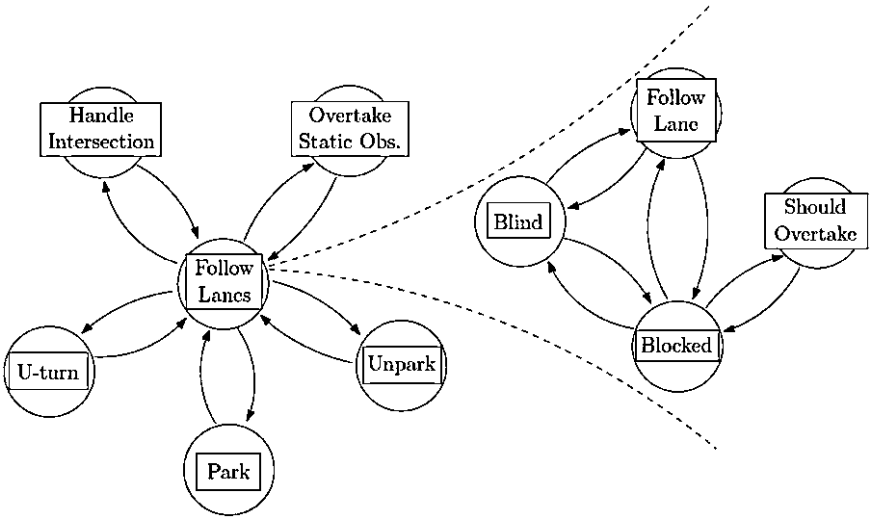


Fig. 11 A portion of the nested finite state automata used to describe urban driving. The automata defined within the `operator-run` state is depicted on the left. Within the `follow-lanes` state is nested an automata that steps through the process of driving along marked roads, shown on the right. This automata is made up of four modes of operation: driving in a lane based on perceptual signals (`follow lane`), falling back on LIDAR/map-based navigation when perceptual signals fail (`blind`), pausing behind another vehicle stopped in the lane (`blocked`) and overtaking said vehicle (`should overtake`)

The only non-trivial of these states is `operator-run`, which corresponds to the operator/user putting the vehicle in an autonomous run-mode. The `operator-run` state is itself expanded into an automata that describes the modes of operation and transitions between these modes. As seen in Fig. 11 the hybrid automaton that corresponds to the `operator-run` mode has the discrete states:

- `Follow-lanes`: navigate along marked roads.
- `Handle-intersection`: navigate through intersections of marked roads.
- `Overtake-static-obstacle`: navigate around an obstacle in the road by driving in the oncoming lane.
- `Execute-u-turn`: turn around on a marked road because of an obstacle completely blocking the road.
- `Park`: navigate into a marked parking spot.
- `Unpark`: navigate out of a marked parking spot.

Each of these modes in turn contain their own hybrid automata.

The `follow-lanes` mode is a hybrid automaton whose discrete states define the modes of operation needed to navigate along marked roads:

- `Follow-lane`: keep the vehicle between the perceived lane markings of the planned road.

- `Blind`: navigate along the road using GPS and LIDAR as the primary sensors. (The vehicle transitions to this state when the vision sensor has failed, e.g. because the vehicle is facing the sun.)
- `Blocked`: come to a stop because of an obstacle in the planned lane.
- `Should-overtake`: get ready to overtake an obstacle blocking the planned lane.

Each of these modes is a primitive discrete state in that they map directly to a mode of operation and do not expand further to into automata. Figure 11 illustrates this portion of the nested finite state automata used to describe the process of urban driving. In this figure, the automata defined within the `operator-run` state is depicted on the left. Within the `follow-lanes` state is nested an automata that steps through the process of driving in a lane based on perceptual signals, including the states `follow-lane`, `blind`, `blocked` and `should-overtake`, shown on the right.

Using the recursive structure of the nested hybrid automata, we were able to represent and manage the complexity of the tasks of the urban challenge. High-level tasks were represented as states in the highest level of the recursive definition. Sub-tasks were represented at lowers levels within each high-level task. This provided a measure of isolation for the definition of these sub-tasks, as transitions between high-level tasks could be defined independently of the state of the sub-tasks below them.

3.7 Steering and Speed Control

Steering and speed control is implemented in a behavior-based voting design [10, 11]. In this design, a number of behaviors evaluate candidate actions over a short temporal scale, each behavior representing a specific interest pertaining to the robot's objective.

In this implementation, as shown in Fig. 12, the behaviors reason over constant curvature arcs. Each behavior distributes an allocation of votes over an array of potential arcs for the robot to navigate along. The behaviors can allocate votes for arcs that work to achieve its interests, or against arcs that are detrimental to its interests. In addition to distributing votes for or against arcs, behaviors assign a maximum allowable velocity, associated with each arc. Behaviors need not necessarily express an interest across both curvature and velocity. A behavior may vote for curvatures and leave the allowable velocities set to the robot's maximum velocity, it may cast no votes for or against curvatures and express its interest across the allowable velocities, or it may express its interest across both dimensions.

To choose a curvature and velocity for the robot to execute, an arbiter sums the votes cast by each behavior for each curvature arc, weighting the votes for each behavior according to a predetermined weighting scheme. It selects for execution

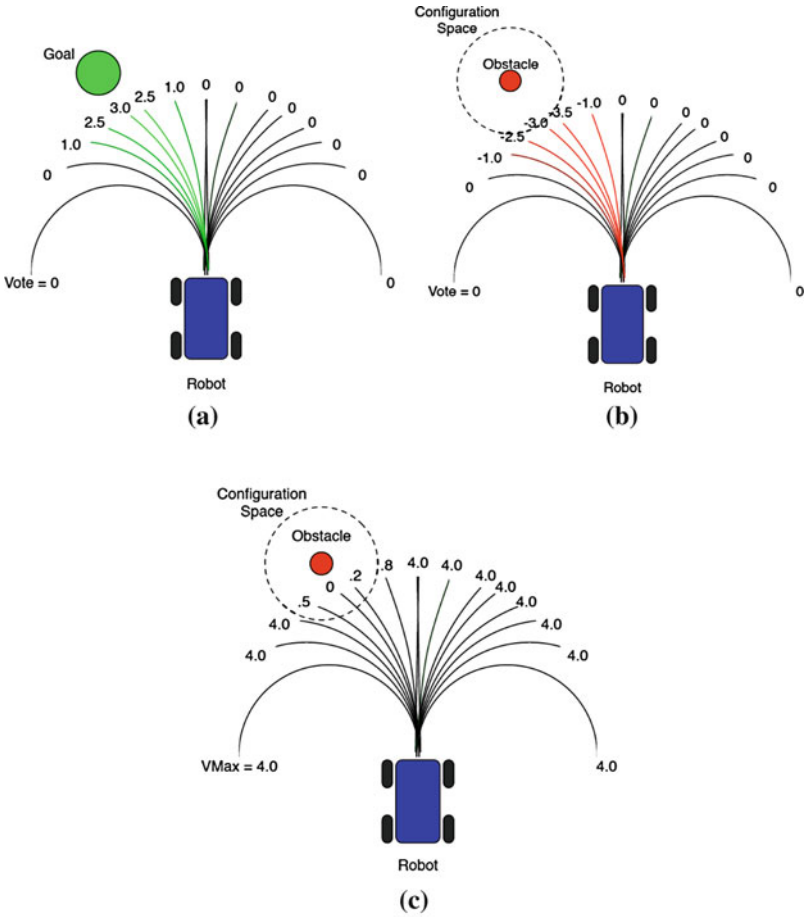


Fig. 12 Three of the behaviors used in the implementation of the reactive layer. In **a**, the *move to waypoint* behavior votes in support of curvatures that take the robot closer to the provided *waypoint*. In **b**, the avoid obstacles behavior votes against curvatures that take the robot toward sensed obstacles. In **c**, the maintain headway behavior sets a maximum allowable translational velocity for each curvature, with respect to sensed obstacles. An arbiter tallies the weighted votes provided by the set of behaviors, and outputs the curvature with the most votes, and the minimum allowable translational velocity for that curvature

the curvature arc with the highest total of votes. It then selects for execution the minimum of the maximum allowable velocities assigned by the respective behaviors to the selected curvature arc. The selected curvature and velocity are sent on to low-level controllers for execution.

Over 30 different behaviors were used in combination to create 25 different control modes. Three of the behaviors are diagrammed in Fig. 12:

Fig. 13 A panoramic photograph of the day-to-day testing site used by sting racing

- *Move to Waypoint*: shown in Fig. 12, allocates positive votes to arcs according to a linear control law relating the local heading to the waypoint to a commanded curvature.
- *Avoid Obstacles*: shown in Fig. 12, allocates negative votes to arcs according to the distance along the arc that the arc crosses into the configuration space around a detected obstacle. Arcs that do not cross into the configuration space of the obstacle are not voted against.
- *Maintain Headway*: shown in Fig. 12, sets maximum allowable velocities for each arc according to the distance along the arc that the arc crosses into the configuration space around a detected obstacle. If the arc does not cross into the configuration space of the obstacle, the robot's maximum speed is assigned. If the arc crosses into the configuration space of the obstacle within a parameterized safety distance, the maximum allowable velocity is zero.

4 Vehicle Performance

In this section, we describe how the system we built performed in our own field trials, during the DARPA site visit, and during the DARPA national qualification event. In general, we found that our system performed well against various challenging scenarios: high speed driving, irregular intersections, road navigation with GPS denial, multi-lane road navigation, *U*-turns on irregular road segments with moving obstacles.

Testing occurred at a handful of locations. Day-to-day testing of primary tasks took place at our on-site facility shown in Fig. 13. Here, we developed algorithms for speed and steering control at up to 25 mph, lane keeping, obstacle field navigation, parking, overtaking road blockages, headway maintenance, and intersection handling. This was also the location used for our DARPA site visit.

Testing of high speed steering and speed control, headway maintenance, lane changing, and high speed obstacle avoidance occurred at the high-speed driver training track at the Georgia Public Safety Training Center (GPSTC) in Forsyth, Georgia. GPSTC is a state-run facility for (among other things) training police

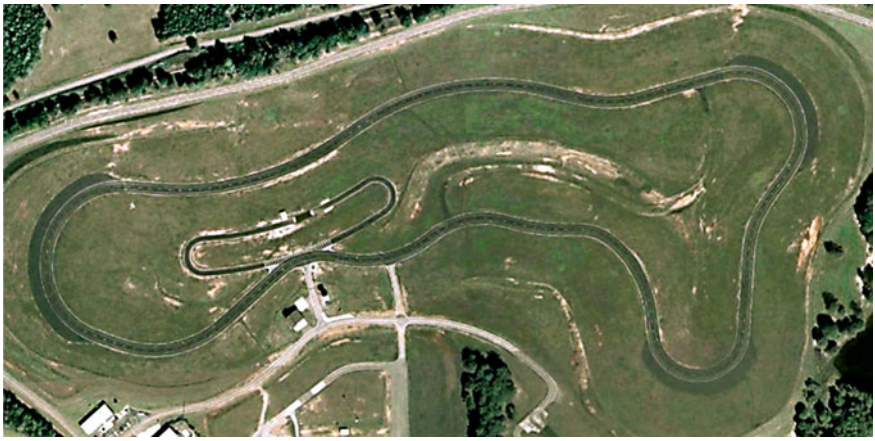


Fig. 14 A satellite image of the Georgia Public Safety Training Center's 1.5 mile high speed test track

officers in high performance, high speed driving both on long road segments, as well as among close-quarters intersections.

This track is shown in Fig. 14. This course was particularly challenging because it was specifically designed to be difficult to handle at high speeds. For example, the roadway has several rolling hills with extraordinarily steep slopes. The road curves counter-intuitively just after the crests of several large hills. In all, this course was ideal for stress testing the robot's ability to handle driving scenarios at higher speeds.

Figure 15 illustrates an example of a test we performed on our robot at the GPSTC high speed track. Figure 15 shows the robot maintaining headway behind another vehicle (in this case, at about 30 mph). As shown in Fig. 15, in a quick motion, the lead vehicle changes lanes, revealing a previously occluded stationary obstacle. If the robot quickly and accurately detects the barrels, the robot should be able to stop prior to collision. (In this case, the left lane may contain oncoming traffic, so the robot must come to a complete stop before changing lanes). Figure 15 shows that the vehicle successfully stopped safely. Subsequently, the robot backed up slightly, and performed an overtake maneuver, and continued on its way.

Also at the GPSTC, we tested on a course specially designed to stress a driver's ability to safely navigate intersections (see Fig. 16). On this course, we developed and evaluated our vehicle's ability to negotiate intersections, with special attention to badly-behaving drivers and pulling out from four-way stop and two-way stop intersections.

In addition to standard driving and intersection handling, the GPSTC urban course allowed us to test other important test scenarios. Several of the intersections are formed from irregularly angled roads. To be successful in this type of scenario, the robot must be robust in its analysis of the presence of other vehicles at an

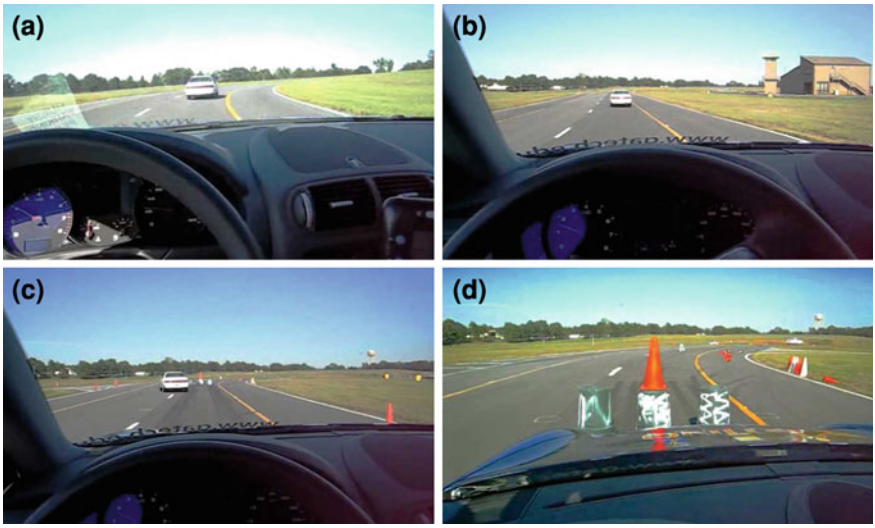


Fig. 15 **a** The robot driving at 25+ mph on the GPSTC high speed track. Taken from safety operator's position. **b** The robot shown following another vehicle at speed. **c** The lead vehicle darts away at the last moment to suddenly expose an occluded stationary object. **d** The robot stops in time before the obstacle

Fig. 16 A satellite image of the GPSTC urban course

intersection (as discussed in Sect. 4). Figure 17 depicts the robot queuing at an intersection behind another vehicle.

4.1 National Qualification Event

The national qualification event was comprised of three courses. Here, we describe the performance of our vehicle at these three courses, as well as draw some conclusions about what could have improved the performance of our vehicle.

Fig. 17 The robot shown queuing behind another vehicle at an intersection on the GPSTC urban course

4.1.1 Course C

Course C at the national qualification event was designed to test a robot's ability to navigate four-way stop intersections, identify road blockages, and execute a *U*-turn. Our robot was able to successfully navigate the outer loop, passing through both intersections. When it approached the road blockage on the southern loop, it properly identified it as such, performed a *U*-turn on the road, and resumed its mission on another route to the active goal point.

4.1.2 Course A

Course A at the national qualification event was constructed to test a robot's ability to handle pulling out from a stop at a two-way intersection. That is, the robot comes to a stop sign at an intersection where traffic on the cross street does not have to stop. We placed our robot at the starting position, initiated its various processes and quickly verified that all systems had kicked off properly. From the point of view of a spectator, when the robot was enabled by the wireless control system, it approached the intersection, slowed momentarily, and collided with the wall on the far side of the intersection. The collision was precipitated by the simultaneous occurrence of two failures.

First, the serial connection between the pose estimation computer and the GPS device hung in a way we did not anticipate. The connection appeared to be alive, but the position information it provided was "stuck" at a fixed point in time. As the car decided to approach the intersection, it increased the input to the accelerator. In reality, it was moving towards the intersection, but because of the error on the GPS serial connection, the robot believed itself to be stationary.

Second, because the robot believed itself to be stationary, the robot calculated that the wall (detected via the LIDAR scanners) had an increasing and positive velocity. Ground returns—points on the ground detected by the LIDAR scanners—had been a common occurrence during our field testing where the ground

was hilly. And because walls have zero velocity and ground returns very often have a high velocity, we had programmed the robot to estimate that what it was seeing was a ground return.

As the other cars on the course approached the robot, their predicted trajectories crossed that of the robot, and the robot attempted to stop as a defensive measure. Unfortunately, this occurred too late for the vehicle to stop in time.

Over many months of testing, we had seen the serial connection to the GPS sensor become livelocked only once before. At the time, we took steps we believed to correct the problem in hardware. As a result of the collision, the front sensor mount was bent, but none of the sensors themselves were damaged at all. The sensor mount was rewelded to the front of the vehicle and the sensors reattached and recalibrated. The next day, we were pleased to see our robot out in testing on the course. Course B at the national qualification event was the largest course, and required the robot exit the starting dock, proceed around the large traffic circle, navigate the road network, and park and unpark from a spot in the parking area.

Our robot successfully navigated onto the traffic circle and made the turn onto the narrow two-lane roadway known as the "chute" that leads out to the larger section of the course. We decided to end the run while the robot was still in the chute because of the apparently confused behavior the robot exhibited.

The chute was constructed by placing *K*-rails right on top of the border of the lanes. This made the road very narrow. We designed the robot to give strict regard to the rules governing vehicle-environment spacing. Specifically, 1 m separation on both sides of the vehicle had to be maintained at all times, while not crossing a double-yellow line. The *K*-rails in the chute were so close to the road as to necessitate the violation of this rule.

When presented with this situation, the robot entered the "Overtake Obstacle" mode, which allows it to cross a double-yellow line in order to get around a blockage. But because the *K*-rail on the opposite side of the road was also so close, the robot was not able to make progress while in overtake mode. In a sense, the robot became "wedged" in the chute. What we failed to anticipate after months and months of making sure that the robot could follow the rules successfully, was that sometimes rules need to be bent a little. Any human driver would easily have identified the chute-setup as a non-regular situation, and would have improvised his way of out the predicament. This, alas, was something our system was ill-equipped to do.

5 Conclusion

Overall, we are happy with many elements of our vehicle's performance. Our system was able to do a number of things effectively, some of which were not tested at the national qualification event. This includes our system's ability to detect and track dynamic obstacles based on the fusion of horizontal planar LIDAR scanners and RADAR scanners. Our robot's ability to drive at full speed

on long, hilly stretches among other vehicles is an accomplishment. The nested hybrid automata approach made development and testing of the complex problem of situational awareness achievable for our small team of researchers.

Acknowledgements We thank the various groups at Georgia Tech who funded most of this work, including the College of Computing, the school of Electrical and Computer engineering, the Georgia Tech Research Institute, and the office of the Vice Provost. Thanks to the Georgia Public Safety Training for use of their facilities for testing our vehicle. Thanks to everyone who contributed to the project. We had very generous support from SAIC in terms of staff and equipment. We would like to thank in particular Robert Schafrik and Karl Kluge from SAIC. In addition, a number of graduate and undergraduate students at Georgia Tech contributed to the project with ideas, early implementations and stress testing of the project.

References

1. Arkin, R.: Behavior-Based Robotics. MIT Press, Cambridge (1992)
2. DARPA: Learning applied to ground robots (lagr) proposer information pamphlet. BAA-04-25 (2004)
3. DARPA: Darpa urban challenge rules and evaluation criteria. http://www.darpa.mil/grandchallenge/rules.asp (2008)
4. Dickmanns, E.D.: Dynamic vision for perception and control of motion. Springer, London (2007)
5. Gat, E., Bonnasso, R.P., Murphy, R., Press, A.: On three-layer architectures. In: Artificial Intelligence and Mobile Robots, pp. 195–210. AAAI Press, Menlo Park (1998)
6. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), pp. 278–292. IEEE Computer Society Press, California (1996)
7. Kennedy, S., Hamilton, J., Martell, H.: Architecture and system performance of span–novatel's gps/ins solution. In: IEEE/ION Position, Location, and Navigation Symposium, San Diego (2006)
8. Kluge, K., Kreucher, C., Lakshmanan, S.: Tracking lane and pavement edges using deformable templates. In: Proceedings of the SPIE, San Jose, vol. 3364, pp. 167–176 (1998)
9. Kluge, K., Thorpe, C.: The yarf system for vision-based road following. In: Mathematical and Computer Modelling, vol. 22, pp. 213–233 (1995)
10. Rosenblatt, J.: Damn: a distributed architecture for mobile navigation. J. Exp. Theor. Artif. Intell. **9**(2), 339–360 (1997)
11. Sun, J., Mehta, T., Wooden, D., Powers, M., Regh, J., Balch, T., Egerstedt, M.: Learning from examples in unstructured, outdoor environments. J. Field Robot. (2006)
12. Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohb, S., Dupont, C., erik Jendrossek, L., Koelen, C., Markey, C., Rummel, C., Niekerk, J.V., Jensen, E., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A., Mahoney, P.: The robot that won the darpa grand challenge. J. Field Robot. **23**, 661–692 (2006)
13. Wooden, D., Powers, M., Egerstedt, M., Christensen, H., Balch, T.: A modular, hybrid system architecture for autonomous, urban driving. J. Aerosp. Comput. Inf. Commun. **4**(12), 1047–1058 (2007)

Development of the NaviGator for the 2007 DARPA Urban Challenge

Carl Crane, David Armstrong, Antonio Arroyo, Antoin Baker, Doug Dankel, Greg Garcia, Nicholas Johnson, Jaesang Lee, Shannon Ridgeway, Eric Schwartz, Eric Thorn, Steve Velat and Ji Hyun Yoon

Abstract This chapter describes the system design developed for Team Gator Nation's submission to the 2007 DARPA Urban Challenge. In this event, vehicles had to navigate on city streets while obeying basic traffic laws. One of the major challenges was interacting with other vehicles such as at intersections. To address these challenges, a hybrid Toyota Highlander was automated and instrumented with pose estimation (GPS and inertial) and object detection (vision and Laser Detection and Ranging) sensors. A control architecture was developed which integrates planning, perception, decision making, and control elements. The intelligence element implements the Adaptive Planning Framework which was developed by researchers at the University of Florida. This framework provides a means for situation assessment, behavior mode evaluation, and behavior selection and execution. This chapter describes this architecture and concludes with lessons learned from participation in the Urban Challenge event.

1 Introduction

In DARPA's vision, "The Urban Challenge features autonomous ground vehicles maneuvering in a mock city environment, executing simulated military supply missions while merging into moving traffic, navigating traffic circles, negotiating busy intersections, and avoiding obstacles." Moving the challenge into an urban setting added structure and complexity to the previous Grand Challenge problem

C. Crane (✉) · D. Armstrong · A. Arroyo · A. Baker · D. Dankel · G. Garcia · N. Johnson · J. Lee · S. Ridgeway · E. Schwartz · E. Thorn · S. Velat · J. H. Yoon
Center for Intelligent Machines and Robotics,
University of Florida, Gainesville, FL 32611, USA
e-mail: carl.crane@gmail.com

D. Armstrong
e-mail: uf.robotics@gmail.com

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_4, © Springer-Verlag London Limited 2012

where success relied on a single mode of operation requiring no interaction with the environment beyond simple traversal. Success in the Urban Challenge necessitated numerous modes of operation and complex interaction with the environment.

The specific problem to be solved is detailed in the Urban Challenge Technical Evaluation Criteria Document [1]. Here the problem was organized into four categories, i.e., Basic Navigation, Basic Traffic, Advanced Navigation, and Advanced Traffic, where each was more complex than the previous. Upon reviewing this document, the authors identified the following set of technical challenges: pavement and lane detection; detection of static obstacles; detection and classification of dynamic objects; environment data representation and sensor integration with noise in sensor systems; localization; high-level mission planning; determination of appropriate behavior mode and smooth transition between modes; and interprocess communication and coordination of multiple threads on multiple computers. Fault tolerance is another obvious concern, but this was only addressed in a limited fashion due to the experimental nature of the vehicle.

Much work has been done in the past 20 years to address many of the technical challenges. Several references [2–7] provide excellent summaries of the advancements made by other teams competing in the 2005 DARPA Grand Challenge. The authors' work related to the 2005 event is published in two references [8, 9]. Numerous additional references can be cited for each of the important technical challenges.

The authors believe that the approach presented here makes new contributions primarily with respect to the determination of the appropriate behavior mode and the smooth transition between modes. Traditional approaches, such as for example vision processing algorithms to identify lane markings in an image, are modified as needed and integrated into the system.

2 Overview of System Architecture

A hybrid Toyota Highlander was selected as the base platform for the system. Steering, throttle, braking, and transmission controls were automated and vision, Laser Detection and Ranging (LADAR), inertial, and GPS sensors were mounted to provide necessary information about the environment. The vehicle system is shown in Fig. 1.

The system architecture is a natural extension of the Joint Architecture for Unmanned Systems (JAUS) Reference Architecture, Version 3.2, which defines a set of reusable components and their interfaces. The actual core software to support the JAUS messaging system was developed and extensively tested for the previous Grand Challenge and supports the current effort with little or no modification required.

At the highest level, the architecture consists of four basic elements, which are depicted in Fig. 2. The planning element contains the components that act as a

Fig. 1 Team Gator Nation NaviGator

repository for a priori data such as the Route Network Definition File (RNDF) which provides the overall database information about the roads, lanes, and intersections, and the Mission Data File (MDF) which provides the set of RNDF waypoints to traverse for a particular mission. This element also performs the high-level route planning and re-planning based on that data plus real-time information provided by the rest of the system. The Control Element contains the Primitive Driver that performs closed-loop control on vehicle actuators to keep the vehicle on a specified path. The Perception Element contains the components that perform the sensing tasks required to determine the vehicle's position, to find a road, to find the lanes on a paved road, to locate both static and dynamic obstacles, and to evaluate the smoothness of terrain. Finally, the Intelligence Element contains the components that work together to determine the best course of action to navigate the vehicle in a complex environment based on the current mission and situation. An overview of a typical sequence of operations of the architecture is presented as follows (reference Fig. 2):

1. The High-Level Planner (HLP) component performs off-line path planning to generate a desired motion path based on the RNDF and the MDF.
2. A tessellated Local World Model (LWM) (300 m $\times$ 300 m grid with 0.5 m resolution) is generated based on the a priori road network data and the planned motion path. The center point of the LWM is located at the current location of the vehicle as determined from sensor positioning data.
3. Data from LADAR and vision sensors, which identify static obstacles, dynamic objects, smooth terrain, and road lane regions, are integrated as a layer into the LWM.

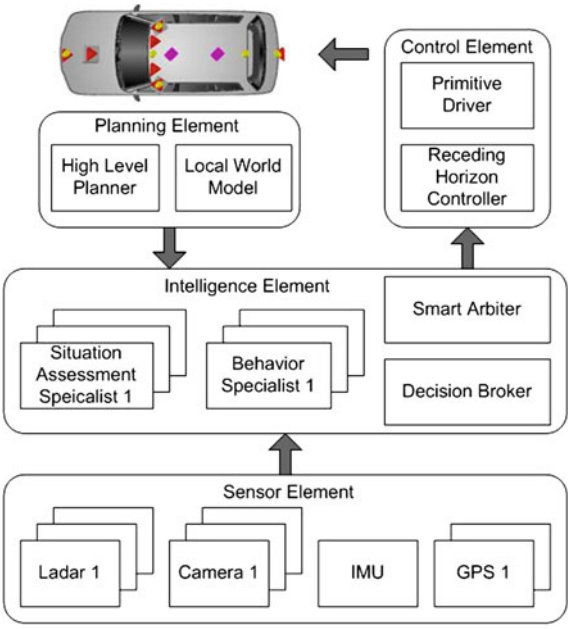


Fig. 2 System architecture

4. Based on the a priori data and sensed data stored in the LWM, software components referred to as Situation Assessment Specialists focus on making specific findings (e.g., one specialist reports if the lane to the left, or right, is clear of other vehicles or obstacles).
5. Seven software components referred to as Behavior Specialists then make an assessment of whether their corresponding behavior mode is appropriate at this moment. The seven behavior modes are Roadway Navigation, Open Area Navigation, Pass Left and Pass Right, Reverse Direction, Intersection Traversal, Off Road, and Parking.
6. A software component referred to as the Decision Broker selects the behavior mode for the system based on the recommendations of the Behavior Specialists.
7. Based on the behavior mode, a software component called the Smart Arbiter generates a 60 m × 60 m traversability grid designed to elicit a specific response from the vehicle (e.g., change lanes).
8. Finally, the Receding Horizon Controller component plans a suitable path through the Smart Arbiter's generated traversability grid. Steering, throttle, and braking commands are generated to execute the planned path.

A description of the components associated with each of the four elements of the architecture, i.e., planning element, sensor element, intelligence element, and control element follows.

3 Planning Element Components

3.1 High-Level Planner

The HLP provides overall guidance to the vehicle. Its functions include:

1. Creating and maintaining a representation of the RNDF that readily allows for efficient data manipulation during route planning,
2. Using the MDF to plan a route through the RNDF representation using an A* algorithm [10],
3. Periodically communicating waypoints to the LWM, so it has an accurate record of the immediate planning space,
4. Re-planning a route when obstacles are encountered, and
5. Collecting data from the sensors about the domain as it is explored and populating the RNDF representation with this information so it contains a more accurate representation of the domain.

3.2 Local World Model

The LWM has multiple roles within the system architecture. First, it generates a model of the world based on the a priori RNDF. It receives a subset of the RNDF waypoints within a 300 m $\times$ 300 m area of the vehicle from the HLP and draws an estimated picture of the world into a rasterized grid using a resolution of 0.5 m. This raster-based approach was chosen because the output from the LWM can then be easily incorporated into other system components. The grid resolution of 0.5 m was chosen from experience in the 2005 DARPA Grand Challenge. Figure 3 shows an example of the 300 m $\times$ 300 m grid. Other components, such as the perception components, which are discussed below, work with a smaller 60 m $\times$ 60 m grid. Any needed information is extracted from the 300 m $\times$ 300 m grid and can be transmitted to any necessary components. Figure 3 shows such a subsampled grid.

After the initial estimate of the world is created from the RNDF, the LWM localizes the vehicle position in the world using data from the Global Positioning (GPOS) component as well as lane-finding and path-finding sensors. The lane-finding and path-finding sensors are incorporated to account for possible discrepancies between the RNDF and the GPOS data. The LWM takes the position of the center of the sensed lane and adjusts the a priori world map to fit the sensed world. Figure 4 gives examples where adjustment is necessary. In this figure, the black (thin) lines represent the actual road as it is sensed relative to the vehicle, the orange (thick) lines are based on the RNDF, and the blue (dark) rectangle signifies the vehicle position which is based on GPOS. In (a), either GPOS is incorrect or the RNDF points are not in the lane. In (b), the waypoints do not describe the road

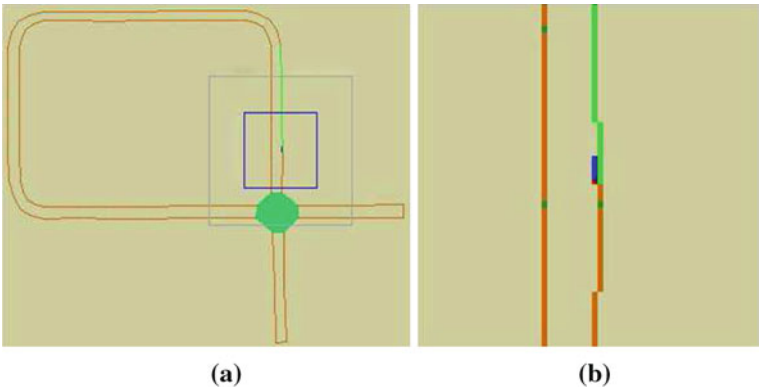


Fig. 3 **a** 300 m × 300 m Raster LWM ; **b** Sub-sampled 60 m × 60 m Grid

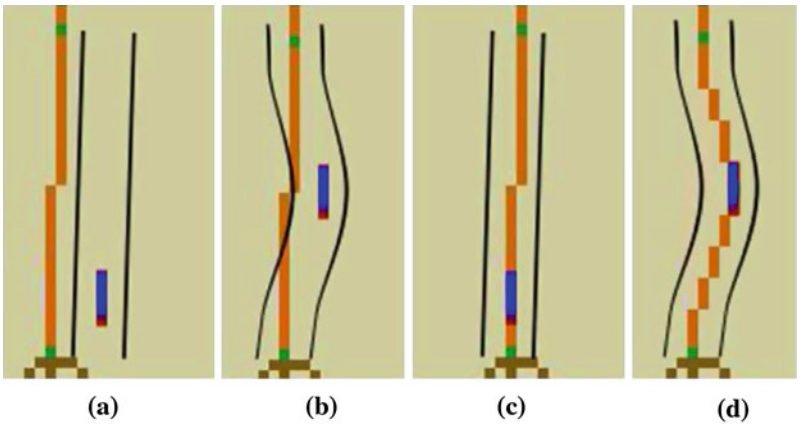


Fig. 4 Arbitration of discrepancy in GPOS Data, RNDF Data, and Sensed Data

accurately. The Local World Model accounts for these errors using data from the lane- and path-finding sensors. In (c), the RNDF map has been shifted to align the RNDF road and the sensed world. In (d), the LWM has added additional waypoints to correct for the discrepancy between the RNDF and the real road.

After localizing the vehicle, the LWM tries to determine important facts about where the vehicle is located in the world and about the vehicle's immediate environment. This information includes whether or not the vehicle is on a road, in an intersection, or in an open area. If the vehicle is on a road, it determines which road the vehicle is on, whether or not there are lanes to the left and right of the current lane, and the direction of travel of those lanes. It also estimates the distance to an intersection, a stop, or an open area. The results of the analysis are shared with the rest of the system through the appropriate Situation Assessment Specialists and Behavior Specialists.

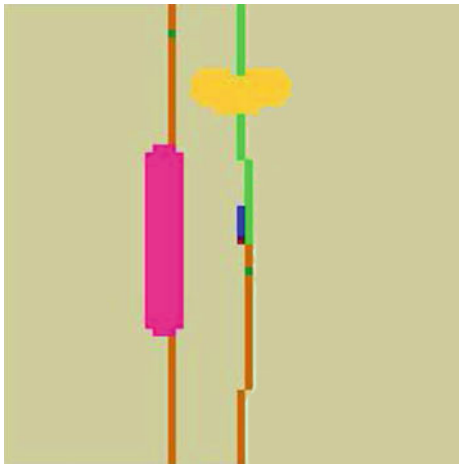


Fig. 5 Moving obstacle depicted in LWM

Next, the LWM is responsible for characterizing, predicting, and injecting dynamic information into the world model, which can then be propagated throughout the system. A list of objects is received from the Moving Object and Classification sensor, which provides the position, velocity, and size of the detected objects. The LWM overlays these objects onto the world map and allows the Urban NaviGator to have a better understanding of what is happening in the world. Figure 5 shows the LWM output with moving objects placed into the world grid. With the moving object information placed into the world grid, the LWM can make estimates on the distance to an object along the road and recommend a speed to the Receding Horizon Controller so that the vehicle will not collide with an object and also maintain a safe separation distance.

Finally, the LWM dynamically spools waypoints to the Receding Horizon Controller. After the HLP has planned a path that completes the mission, it provides a rough plan to the Local World Model that contains only the checkpoints, entry points, and exit points that need to be traversed. The LWM then takes the rough plan and fills in the intermediate waypoints that need to be traversed to travel from one HLP point to another. This provides the flexibility to modify the waypoints that need to be traversed based upon the current operating behavior without re-planning the overall mission. Figure 6 shows a change in the mission waypoints based upon a change in the operating behavior. In (a) all the mission points sent by the HLP are shown. This mission involves making a loop around the course and coming to a stop at the segment at the bottom. In (b) a number of intermediate points have been filled in to be sent to the Smart Arbiter component (discussed subsequently). All points up to a set distance away from the vehicle are sent. In (c) the mission points have been shifted to the other (left) lane to execute a change lane behavior due to an obstacle (represented by the small solid blue rectangle near the bottom of the right lane) detected in the same lane.

In summary, the LWM provides a detailed 300 m $\times$ 300 m representation of the environment around the vehicle. It receives a priori roadway data from the

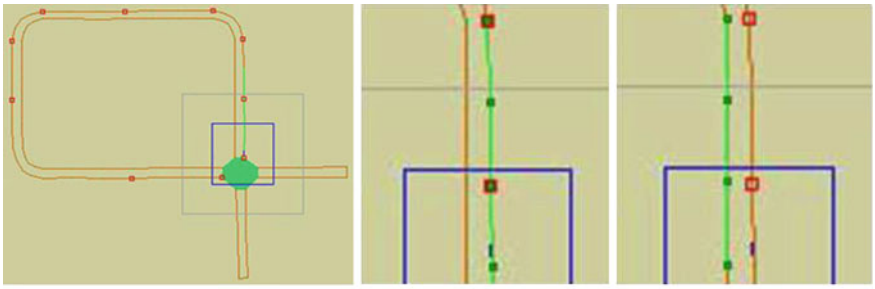


Fig. 6 Dynamic update of intermediate points

HLP as well as static and dynamic obstacle and lane information from the perception system. The LWM constantly estimates any discrepancies between the a priori and sensed data by calculating a net offset that can be applied to the a priori data in the event that sensed data is momentarily lost. It determines important facts about the vehicle's position in the world to help make decisions on the appropriateness of certain behaviors. Lastly, the Local World Model maintains a list of mission goal points identifying the correct lane of travel. This information is transmitted to the Smart Arbiter component for motion execution.

4 Sensor Element Components

4.1 Localization

Geo-localization is achieved using a GE Aviation North Finding Module (NFM) combined with two GPS units and one odometer. The NFM is an inertial navigation system that maintains Kalman Filter estimates of the vehicle's global position and orientation as well as angular and linear velocities.

The system design is predicated on redundancy and relative accuracy when GPS is lost. The GPS signal provided to the NFM comes from one of the two onboard GPS units. They include a NovAtel Propak-V3-HP with OmniSTAR subscription service, and a Garmin WAAS Enabled GPS 16. An onboard computer simultaneously parses data from the two GPS units and routes the best-determined signal to the NFM. This is done to maintain the best available GPS information to the NFM at all times. The best GPS solution is determined by evaluating each signal with respect to its unit type, mode of operation, Horizontal Dilution of Precision (HDOP) value, Root Mean Square (RMS) error, number of satellites, and duration of uninterrupted signal, among other criteria. The NFM has been programmed to use a different set of tuning parameters in its Kalman Filter depending on what type of GPS signal it is receiving.

Fig. 7 **a** GPOS repeatability data; **b** Magnified view

In the event that both GPS units lose track of satellites, as seen during GPS outages such as when the vehicle is in a tunnel, the NFM will maintain localization estimates based on inertial and odometer data. This allows the vehicle to continue on course for a period of time; however, the solution will gradually drift and the accuracy of GPOS will steadily decrease as long as the GPS outage continues. Under ideal conditions the GPOS system typically maintains Global position accuracies and repeatability in the range of 0.1–0.5 m. Figure 7 shows five laps around a 0.6 mile test track with GPS (blue lines) and five laps without GPS (red lines). The vehicle was driven as close as possible in the center of the road (road edges are 28 ft apart and are marked by green lines) for every lap. Without GPS, the NFM was using only the encoder signals to damp the velocity errors. Under these conditions the GPOS system maintains Global position accuracies less than 5 meters for a distance traveled of approximately 3 miles without GPS.

4.2 Perception

The sensor packaged deployed on the vehicle includes an array of LADAR and vision sensors. These include six SICK LMS-291 LADARs, two SICK LD-LRS1000 long-range LADARs, and six Matrix Vision BlueFOX high-speed USB2.0 color cameras. Moreover, many of the sensors deployed on the vehicle are articulated with one degree of freedom. Figure 8 depicts the sensor configuration.

4.2.1 Smart Sensor Concept

The concept of a *Smart Sensor* is to take various sensor inputs and unify them into a generalized format that can be directly combined. To accomplish this broad task, a generalized data representation was designed that serves to unify the generation, transfer, and analysis of sensor information. This representation, known as the *traversability grid,* consists of a tessellated grid tied to a geo-spatial location at the time of generation. The grid then serves as the base data structure for all arbitration

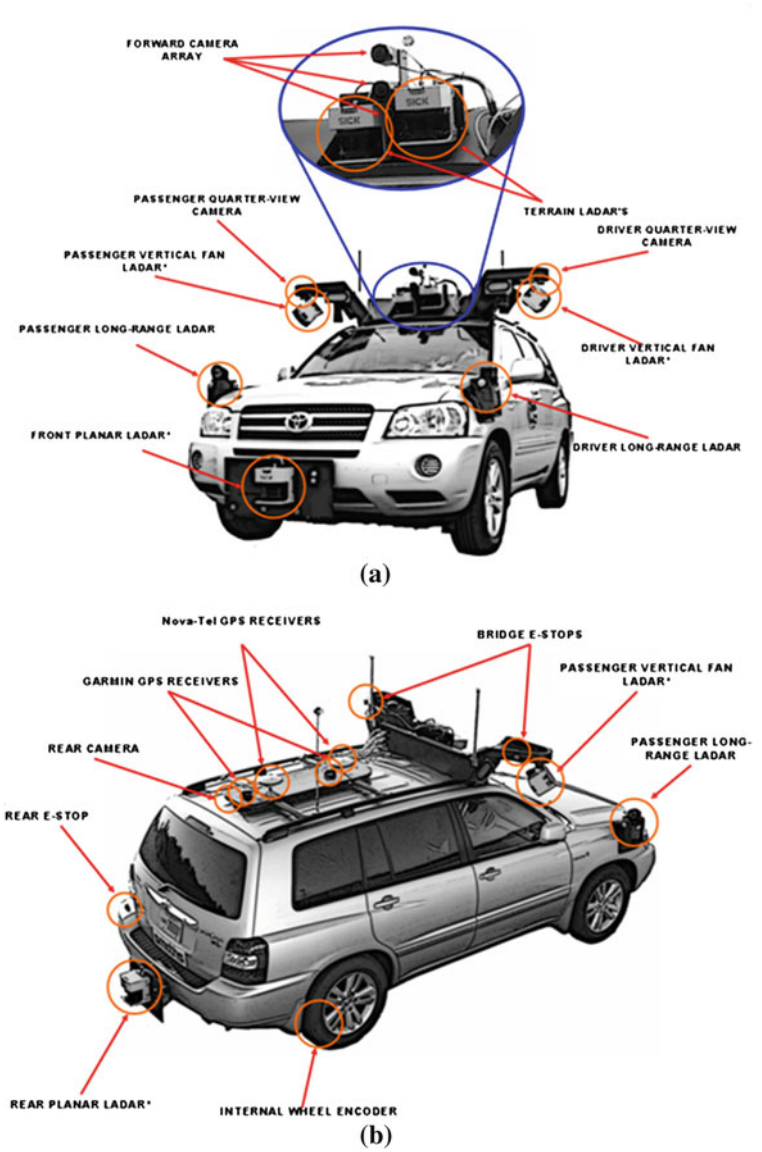


Fig. 8 Sensor configuration

processes and can easily be spatially added and fused with other data-sources which are also in a generalized grid representation. Figure 9 depicts three example traversability grids and the result of sensor fusion by an arbitration component.

By utilizing a common data representation, developers can work independent of arbitration components. Moreover, the Smart-Sensing components can operate asynchronously at a wide variety of rates and grid resolutions due to the spatially

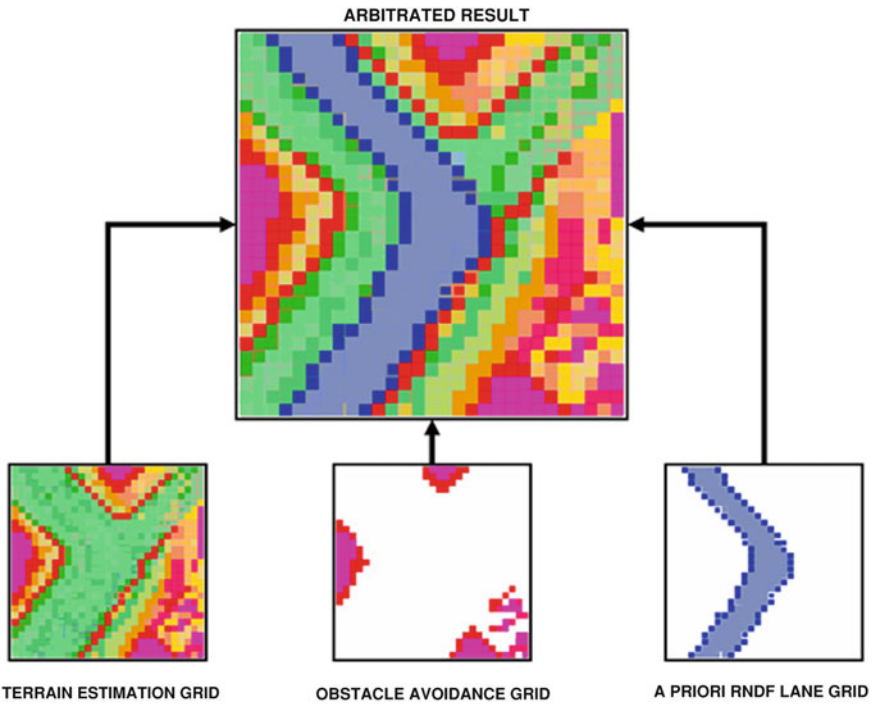


Fig. 9 Smart sensor traversability grid fusion

located nature of the traversability grid [11]. This is possible due to the spatial mapping of each grid as it is fused with the other available sensor information. Thus, the arbitration process takes into account the geo-spatial offsets between the various Smart-Sensor traversability grids when fusing their information so the resulting representation is consistent as the vehicle moves regardless of speed, orientation, or position.

4.2.2 Sensor Groups

The main objectives of the perception systems are to characterize terrain, localize moving and stationary obstacles, and provide error information about the vehicle's pose in the lane. To this end, the development of the sensing systems was split into three main thrusts. The first concerned terrain characterization. For this, a combination of vision and LADAR sensors were used to look at the slope, relative height, and regularity of texture of the terrain around the vehicle. The second involved obstacle localization, specifically Moving Obstacles (MOs). Since the vehicle is intended to operate with other moving vehicles at speeds up to 30 mph, it is necessary to use sensor data with the largest available range to detect and track potential MOs. To this end, the LD-LRS1000 LADAR sensors on the front fenders

were relied upon to provide obstacle data out to 275 m. This long-range data was supplemented by the more common close-range LMS-291 LADAR sensors on the front and rear articulated bumper mounts. The last thrust of the sensor package was to characterize the road and to determine the pose of the vehicle within the lane. This process was necessarily heavily vision-based. A series of cameras located on the main sensor bridge both in the center and on each of the wings provided high frame rate images of the area in front of the vehicle and were used to isolate the visual elements representing the lane demarcations. From the fusion of this information with some supplemental data from the vertical-fan LADARs, it was possible to estimate the error between the vehicle's current position and the center of the lane. The following sections detail the operation of these three main areas.

4.2.3 Terrain Characterization

The process employed for characterizing terrain was fairly well-defined in the previous Grand Challenges. To this end, the Urban NaviGator utilizes a series of planar oriented articulated LMS-291 LADARs as well as a series of statically mounted LMS-291 LADARs oriented to give scan lines in front of the vehicle at 10 and 20 m, respectively. The point-cloud generated by the sensors is then processed into a tessellated grid representation in which each grid element is processed to generate a relative height, slope, and best-fit plane. The data gathered by these LADARs is then fused with vision-based Path-Finder information. The Path-Finder is a component that completes a color and texture based analysis of the scene in the image to determine what type of terrain the vehicle is on and the relative location of other similar terrain. The result is a continuous and cohesive representation of the terrain surrounding the vehicle which is used to populate a traversability grid.

4.2.4 Moving Obstacle Detection

This area of development is probably the most critical for success in the Urban Challenge. The largest deviation from the previous competitions is the addition of manned and unmanned moving traffic, which represents an entirely new and complex series of problems for unmanned navigation. The first major hurdle is identifying the moving obstacle from the surrounding terrain. This task is made more difficult by the lack of uniformity among competing vehicles both in size, shape, and operating speeds. Furthermore, the various scenarios prescribed by DARPA implied that a high degree of resolution would be required to adequately track vehicles and provide for persistence through occlusions and other events. To this end, the MO characterization proceeded by developing LADAR-based components that could sample the point-cloud of data at a given time-period and generate a cluster of data points that were linearly related. Each cluster then became a candidate for a moving obstacle. Through successive time-steps the

velocity state of each candidate object was re-evaluated and corrected for the movement of the vehicle's reference frame. The result was the effective localization and tracking of every significant static and moving obstacle within sensor range. The list of such obstacles was then provided to the LWM component, which would decide if the obstacles represented other vehicles on the road-network defined in the RNDF.

4.2.5 Vehicle Localization

The final development area is that of localizing the vehicle within the lane on a given road. To do so, a series of cameras are used to sample images from around the front and sides of the vehicle as well as the vertically-oriented LADARs mounted on the wings of the sensor bridge. The captured images are then processed to extract color and edge information and the LADAR data for curb detection. The results of each of these individual processes are then sent to an arbitration component to fuse the various corrections and generate a single set of so-called 'breadcrumbs' to correct the local world model and maintain the vehicle well-posed within the lane.

Vision-based Lane-Finder Smart Sensor

The Lane-Finder Smart Sensor (LFSS) is a vision-based component that utilizes color processing and edge detection using the Hough Transform to isolate linear elements in images captured from the front and wing located cameras. These elements are then sorted on both color and spatial properties and clustered to generate a list of lane-demarcation candidates. Each subsequent candidate group is then evaluated to determine which (if any) bound the vehicle at the current time. If any are found, then the entities are projected into the vehicle reference frame and the relative orientation of the lane is determined at various distances from the vehicle (5, 10, 20 meters). These corrections are then sent to the Lane Correction Arbiter for sensor fusion.

Vision-based Path-Finder Smart Sensor

The Vision-based Path-Finder Smart Sensor (VPFSS) is another vision-based system which attempts to isolate not the lane demarcations, but the boundaries of the drivable road. It accomplishes this task by using color and textural analysis based on the Expectation Maximization algorithm to estimate which areas of the image represent road or non-road. The results of the process are then projected into the vehicle reference frame in the form of a traversability grid, are directly fused with the terrain evaluation results discussed earlier, and allow the generation of an

estimate of the lateral offset of the vehicle from the edge of the road. These results are then forwarded to the Lane Correction Arbiter for sensor fusion.

LADAR-based Path-Finder Smart Sensor

The LADAR-based Path-Finder Smart Sensor (LPFSS) is a LADAR-based component that relies on the passenger-side vertically-oriented LADAR. This LADAR is actuated, allowing it to rotate about an axis normal to the ground plane and is oriented such that the beam sweeps an arc normal to the ground plane. Moreover, the actuation of the LADAR is behavior based. Among other behaviors, the actuator will, in normal roadway navigation, attempt to orient the LADAR to track the road-boundary adjacent to the rear axle. The result of processing the LADAR scans is an estimate of the lateral offset from the rear passenger wheel to the edge of the road and serves as a good baseline for all other corrections. Again, the results of this component are forwarded to the Lane Correction Arbiter for sensor fusion.

Lane Correction Arbiter Smart Sensor

The Lane Correction Arbiter is a pseudo smart-sensor in which it does not originate any new sensor data through acquisition from sensor hardware; rather it processes other smart-sensor results into a further refined solution. The purpose of the Lane Correction Arbiter Smart Sensor (LCASS) is to filter, persist, and decay the various lane correction estimates of the other vehicle localization smart sensors into a single "best-fit" solution. To do so, the arbiter relies upon an internal vehicle reference traversability grid that is maintained over time and holds the relative lane-center estimates provided by the contributing smart sensors. Each lane-center estimate is given an initial weight and added to a volatile list of estimates. At each time-step, the content of the list, which contains both the relative position of the lane-center and its current value, is translated and rotated to the inverse of the motion the vehicle frame. In effect, the result is a truly local grid representation of the derived sensor information. The weighted entities depicted in the grid are decayed over time based on their position, the velocity of the vehicle, and the originator of their data. The resulting grid is then used to drive a variable order curve fit (ranging up to fourth order), which attempts to minimize the variance of the fitted curve at regular intervals. Once the curve is generated, a series of vehicle referenced lane correction offsets are generated for 0, 5, 10, 15, and 25 m.

5 Intelligence Element Components

Team Gator Nation has developed and deployed the Adaptive Planning Framework [19] to address the issues associated with behavior mode selection in complex or unstructured environments presented during the DARPA Urban Challenge.

It enables the vehicle to intelligently select the most appropriate behavioral characteristics given the perceived operating environment. The framework is scalable to systems of varying complexity and size and is compatible with existing architectures such as JAUS RA-3.2, NIST 4D/RCS, and others. The adaptive planning framework is composed of three principle elements tasked with assessing the situation, determining the suitability and viability of all possible solutions, and executing the most suitable of all recommended solutions. These three component types are multiple Situation Assessment components, multiple Behavior Specialists, and one Decision Broker component.

5.1 Situation Assessment Specialist Components

Dynamic environment information, originating from any array of sensors is monitored and managed by the Situation Assessment Specialists. Each specialist design is tailored to the sensor or collection of sensors whose data it will be analyzing. While the inputs to the specialist can come from any data source, the output or "finding" must adhere to specific guidelines outlined by the framework. Findings can be in the form of conditions, state, or events. Once the findings have been generated the information is disseminated to all other components that might need it. An example of a situation assessment specialist would be a software component whose sole function was to determine if it is safe to move to the adjacent lane. This component would monitor sensor data as reported by the Moving Objects sensor and reach a Boolean conclusion that would be stored as metadata for use by other processes.

5.2 Behavior Specialist Components

The findings rendered by the Situation Assessment Specialists are consumed by the Behavior Specialists. There is a one-to-one mapping of each behavior with a Behavior Specialist. The role of each specialist is to monitor the findings and evaluate the suitability of its behavior under the current perceived operating conditions. An example of a behavior specialist is the Pass Left/Right behavior specialist. This specialist simultaneously monitors the desired travel lane, as well as adjacent travel lanes, for obstructions. Based on all the inputs, the specialist recommends whether or not a lane change is an appropriate and safe option. As with the specialist findings, the default recommendation is unsuitable and must be proven appropriate at every iteration of the program to ensure truth of the results and operating safety. The Behavior Specialists do not possess the ability to activate or deactivate their associated behavior; such authority is only given to the decision broker.

5.3 Decision-Broker Component

At the highest level of the framework lies the decision broker. Its role is to monitor all Behavior specialist recommendations. It assumes ultimate authority over how the Urban NaviGator will operate while in autonomous mode. Like the other entities within the framework the decision broker can base its conclusions not only on the recommendations and findings of other specialists, but also on data from any other pertinent source. The decision broker employs an asynchronous, iterative, forward chaining reasoning approach to decision-making. With each iteration, findings are evaluated assuming an accessible environment and a list of appropriate behaviors is generated. Operating behaviors are selected using a decision tree approach where preferred operator behaviors are given favorable ordering at higher ply levels. The deployed implementation of the Adaptive Planning Framework centralizes all the decision broker functionality within the JAUS Subsystem Commander and has the added responsibility of selecting which component receives control of the vehicle's JAUS Primitive Driver.

5.4 Behaviors Used During the Urban Challenge

The Urban NaviGator is programed with seven operating behavior modes where each behavior is comprised of a series of sub-behavior modes. An operating behavior is a high-level classification of numerous actions that are harmoniously executed by the various planning, control, and perception elements onboard the UGV to achieve a common operating profile. Each operating behavior state can further be decomposed into more generic and atomic constituent sub-behavior states. This logical organization affords the design of protocols for allowable transitioning amongst behaviors and sub-behaviors to ensure continuity of control and robustness to errors. Some sub-behaviors may be optional, depending on the mission plan or ambient conditions. Vehicle performance is denoted by a sub-behavior status indicator. A failure protocol is incorporated into each sub-behavior should sufficient environmental changes warrant the current vehicle operation inappropriate or unsafe. For example, the Pass-left operating behavior consists of three sub-behaviors. The three atomic sub-behaviors are lane-change left, overtake vehicle, and lane-change right. Successful completion of the Pass-left sub-behavior requires transition through all three sub-behaviors. Only then is a transition back to Roadway Navigation behavior allowed. Should the Pass-left behavior specialist find the behavior to be unsuitable then the a priori defined failure protocol for the given behavior/sub-behavior combination would execute. In the case of the Pass-left operating behavior and sub-behavior overtake, the abort protocol would reduce the vehicle speed, and then attempt a lane-change right as soon as safely possible. In most cases the vehicle is able to recover to a default safe operational state. However, in some cases, such as a catastrophic

system failure or an excessively hostile environment, the safest course of action is for the vehicle to pause and wait for more favorable conditions. The corresponding behavior specialist constantly evaluates the appropriateness of its behavior mode and the decision broker determines which mode will operate the vehicle. The seven behavior modes are described subsequently.

5.4.1 Roadway Navigation

The Roadway Navigation behavior is the primary driving behavior deriving commands to be sent to the vehicle actuators while the objective is lane following. This behavior allows the vehicle to navigate the roadway within the lines of its desired lane. The default sub-behavior is to maintain a safe following distance behind any vehicles ahead. Other sub-behaviors include lane changes on a multi-lane road in order to pass through a mission goal point.

5.4.2 Open-Area Navigation

Open-area navigation is a behavior that should only be used in special circumstances during the Urban Challenge event. This behavior allows the vehicle to move toward a goal location without striking any object, while avoiding any rough terrain. This is in effect the only behavior mode that was required in the 2005 DARPA Urban Challenge. It is useful in the Urban Challenge when the vehicle is in an open area such as a parking lot or an obstacle field. The associated sub-behaviors are Enter Open Area, Exit Open Area, Enter Parking Space, and Exit Parking Space. Thus if the mission plan for the open area does not contain parking spaces the system would transition from the Enter Open Area to the Exit Open Area sub-behavior.

5.4.3 Pass Left and Pass Right

The Pass-Left and Pass-Right maneuvers are used in passing situations where a static obstruction impedes progress in the desired lane but there exists an adjacent available travel lane. Successful Pass Left Behavior execution entails a Lane Change Left sub-behavior, Passing Vehicle sub-behavior, and Lane Change Right sub-behavior. This behavior implies a momentary lane change for obstacle avoidance.

5.4.4 Reverse Direction

This behavior is called whenever it is determined that the current lane is blocked and there is no alternate clear lane available for passing. It is also applicable in cases where the vehicle has entered a ‘dead end’ road that it must ‘escape’ to reach

a mission goal point. The default sub-behavior is to execute an N-point turn sub-behavior protocol.

5.4.5 Intersection Traversal

The intersection traversal behavior is applicable when the vehicle enters the vicinity of an intersection. This is one of the most complicated behavior modes in that the system must rely on a series of Situation Assessment Specialists to safely navigate the intersection. This behavior mode must handle queuing, stopping at the stop line, determining right of way, and ultimately traveling through the intersection while avoiding other vehicles. These steps are compartmentalized into five sub-behaviors: queue to intersection, stop at intersection, queue turn, clear intersection, and finally traverse intersection. It should be noted that if there is no stop at the intersection the sub-behavior will transition from queue to intersection to queue Turn.

5.4.6 Off Road

This behavior is called when a sparse way-point problem is identified or when the MDF indicates an unmarked or dirt road. The default sub-behavior is defensive/reflexive. In this sub-behavior the vehicle operates in a heightened state of cautiousness. The subsystem commander enforces more stringent speed limits based on inertial measurement sensor feedback, other perception algorithms are returned for path finding as opposed to lane finding and line following, and sensor grid maps are arbitrated to give more freedom to the A-star vehicle path planner for reflexive obstacle avoidance.

5.4.7 Parking

This behavior must deal with the problems that arise in the parking lot scenario where precise motion is necessary. When the vehicle approaches the vicinity of an assigned parking space, precise path planning will be initiated to align the vehicle as required. Situation Assessment Specialists monitor the near surroundings of the vehicle to center the vehicle in its parking space while avoiding any static or dynamic objects.

5.5 Smart Arbiter Component

The purpose of the Smart Arbiter component is to generate a 60 m $\times$ 60 m traversability grid, centered at the vehicle's current position, which is used to implement a desired behavior. Motion execution, which is discussed in the next

section, is accomplished via an A* search through this grid to determine the least cost path. In most cases, the least cost path will be obvious as the grid has been constructed to accomplish a desired action. An important feature of this entire approach is that specific behavior modes can be changed with smooth continual control of the vehicle. The Smart Arbiter obtains inputs from the Terrain Smart Sensor, the Lane Finding Smart Sensor, the Path Finding Smart Sensor, and the Local World Model and builds its grid based on the current behavior mode of the system. For example, if the system is in the Roadway Navigation behavior, then the grid cells corresponding to the positions of the line on the edge of the lane, as identified by the Lane Finding Smart Sensor, will be marked as non-traversable regions in the Smart Arbiter grid. The cells corresponding to the road lane will be marked as highly traversable. This will prevent the planner from planning outside the current lane.

6 Control Element Components

6.1 Receding Horizon Controller Component

Low-level path planning on the Urban NaviGator for the basic road following, intersection, passing, and open-area behaviors was facilitated by combining an A* search algorithm with a Receding Horizon Control (RHC) strategy. This made use of a modified form of Model Predictive Control [20, 21]. This procedure expanded a state-space search tree through the arbitrated traversability grid. The search nodes in the A* algorithm were expanded using a vehicle kinematic model. This model took into account vehicle-specific parameters such as turning rate and steering angle per percent effort, where percent effort was the unit of actuation, and used the following kinematic equations of motion to predict the vehicle's location at some future time for a desired steering effort and assuming a constant speed:

$$\Delta x = \frac{1}{k[\cos\phi(1-\cos(\Delta\phi)) - \sin(\Delta\phi)]} \tag{1}$$

$$\Delta y = \frac{1}{k[\cos\phi\sin(\Delta\phi) + \sin\phi(1-\cos(\Delta\phi))]} \tag{2}$$

where ϕ is the current heading of the vehicle, $\Delta\phi$ the predicted change in heading, and k is the radius of curvature of the path the vehicle followed for a particular desired steering effort.

The objective is to find the optimal trajectory that minimized the cost of traversal through the search tree to the goal state. The cost of traversing an arc in the tree to a destination node was composed of costs associated with the traversability values of the cells the arc passed through and costs associated with the offset of a particular destination node from the perceived center of the desired lane and from

the goal state. The expanded search nodes were placed in a priority queue that sorted them based on cost, with the lowest cost node always at the top of the queue waiting to be expanded next. As a node was expanded, it was linked to all its predecessor nodes, allowing the algorithm to keep track of the entire solution path once a search node fell within the goal region. The goal points were calculated from a list of path segments constructed by the HLP and LWM, and were updated, as is customary in RHC, to keep the goal point on the edge of the grid at the motion planner's time-horizon.

The algorithm was modified by feeding the vehicle position forward in time to account for a small system lag. This was accomplished by passing the vehicle position through the aforementioned vehicle model at a time in the future equivalent to the lag time. The first generation of the search was expanded such that the previous steering command was always included as the central node to ensure that the initial steering effort of the previous optimal solution path was an option for the next iteration of the search algorithm. Another special feature of the motion planner was the ability to plan forward, while the vehicle transmission was in drive, and backward, while in reverse using the inverted vehicle kinematics. This was utilized when backing out of a parking space or backing away from a roadblock.

Once the optimal path was found, the trajectory was traced back to the first state change to find the initial steering command to be sent to the steering actuator. The output steering command was in the form of a positive or negative propulsive rotational effort. Closed-loop steering control was accomplished by repeating the algorithm at approximately 40 Hz with position and actuation feedback. If the algorithm failed to find a solution path in this amount of time, the fallback was to slow down and repeat the search with the advantage that as the vehicle traveled more slowly, it was more maneuverable and more likely to find a solution. The vehicle continued to slow down to a stop if necessary if the algorithm continually failed to find a solution.

Figure 10 shows an output traversability grid that was generated by the motion planning component during the Roadway Navigation behavior. The vehicle position is at the center of the grid and the vehicle is heading toward the left as the vehicle attempts to navigate an intersection for a left-hand turn onto an adjacent road segment. Low-cost cells, which identify the lane, are colored blue and purple (line going from the bottom to the right), while non-traversable cells are colored red (background) and yellow (round spots) and assigned high costs. Potential paths that were considered are painted brown (jagged areas), and the solution trajectory is painted in light blue and follows the center of the desired lane of travel.

The speed of the Urban NaviGator was maintained using a proportional-integral-derivative (PID) controller. The controller selected the desired speed as the lowest of several recommendations coming from the path segment list based on the RNDF and MDF speed limits, from the Subsystem Commander based on the current behavior and perceived driving conditions, and from the LWM based on proximity to intersections and moving or static obstacles in the lane. The controller could further limit the speed based on path curvature and traversability of the

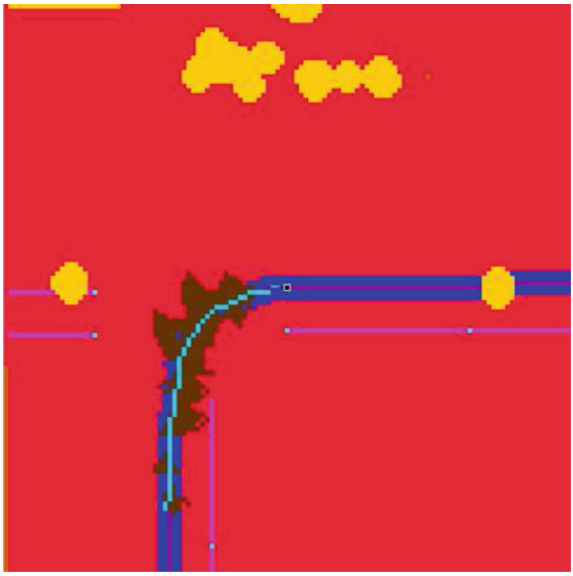


Fig. 10 Output traversability grid showing the potential trajectories the motion planner explored and the solution trajectory

chosen path. A speed command, s, was then calculated using appropriate acceleration or deceleration values specific to the Urban NaviGator and was passed to the controller. The output of the controller was in the form of a propulsive or resistive linear effort represented by

$$\text{linear effort} = K_{FF}s + \text{Bias}_{FF} + K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de}{dt} \qquad (3)$$

where $e(t)$ is the error between the desired speed and the actual speed. Again, the control loop was closed by repeating the algorithm at approximately 40 Hz using feedback from the Velocity State Sensor. The rotational and linear efforts were then sent to the primitive driver component (PD), which converted the efforts to hardware commands and passed them to the actuators.

6.2 Primitive Driver Component

The primitive driver component closed the loop between the motion planning component and the vehicle. The wrench commands and shifter commands generated by the local path planner and the PID speed controller were converted from their native values to hardware-specific values. The steering and shifter commands were converted to absolute motor positions for the motors connected to the steering column and shifting mechanism. The motors had an internal PID controller which was tuned to accurately achieve required positions in a timely manner. Throttle and brake wrench commands were converted to voltages, which

were fed into the Urban NaviGator's pre-existing throttle and brake drive-by-wire electronic control units (ECUs). The PD was also responsible for monitoring the states of all the motors and kill-switch. Lastly, the PD collected feedback from the motors and ECUs, and reported the implemented control back to the motion planner.

7 Results and Lessons Learned

The performance of the implemented architecture at the DARPA Urban Challenge was mostly satisfactory, but less than desired with respect to certain scenarios. The system performed most subtasks well, but failed to fully realize the potential of the design. The qualification event was comprised of missions planned on three courses. The vehicle ran on all three courses with some success.

The adaptive planning framework correctly managed the system's behavior with respect to the sensed scenario. Low-level control of the vehicle was maintained during imposed behaviors by the architecture, leading to smooth continuous driving behavior.

Course A exposed a deficiency in the persistence of moving objects in the implementation. This course simulated a two-way traffic circle that the autonomous vehicle had to merge into and out of. Sometimes traffic vehicles became occluded by others, leading the autonomous vehicle to incorrectly determine it had right of way and could proceed. An attempt to tune this deficiency's effect down was not successful, mostly due to the lack of testing.

Course B exposed an error in the search methodology for the open-area behavior. This test involved navigating through a large open area to a road network to complete a mission. The search space the algorithm considered for the open area was uniform in costs related to traversability, leading to an ill-conditioned optimization problem. This situation was not handled, and the component controlling the vehicle became non responsive.

Courses A and C exposed a "ground strike" problem with moving object detection. Course C was designed to test intersection precedence and re-planning. In both A and C, ground strikes from the LADAR sensors were detected as fixed objects that had to be considered in the intersection and roadway navigation behaviors. These false detections lead to less than desirable behaviors for the scenarios encountered.

The LCASS concept provided accurate information concerning lane center relative to the vehicle location. The utilization of this information in a grid resolution of 0.5 m proved to be problematic. Typical lane widths encountered at the DARPA Urban Challenge site were 3.5–4 m. Typical lane center corrections were often smaller than the grid resolution, leading to situations where the vehicle left the lane due to the lack of precision in lane representation. These problems were observed in roadway navigation on lanes less than 5 m.

Most failures observed involved component implementation errors. The overall architecture worked as designed, given the performance of the components. The implementation does not have significant simulation capacities. Testing was performed with the system deployed in a suitable environment. The development of hardware in the loop simulation for the system could have allowed many of the shortcomings of the implementations to be identified and fixed in a shorter time.

8 Conclusions

The performance requirements identified in the Urban Challenge Technical Evaluation Criteria were challenging. The system had to be able to detect and model its environment and then plan and execute appropriate actions in real-time.

The approach described in this chapter was generated after careful consideration of the design requirements. The central concept is the integration of a priori and sensed information in a raster format in the LWM. Based on this information, an appropriate behavior is selected via arbitration. The behavior is executed by generation of a navigation grid coupled with metadata.

The primary new contribution of this approach involved solving the technical challenges of (a) the determination of the appropriate behavior mode, and (b) the smooth transition of vehicle control between behavior modes.

Acknowledgments The members of Team Gator Nation would like to gratefully acknowledge the sponsorship and support of GE Aviation, the Air Force Research Laboratory at Tyndall Air Force Base, and the University of Florida.

References

1. DARPA: Urban Challenge Technical Evaluation Criteria, 16 March (2006)
2. Urmson, C., et al.: A robust approach to high-speed navigation for unrehearsed Dessert Terrain. J. Field. Robot. **23**(8), 467–508 (2006)
3. Trepagnier, P., et al.: KAT-5: robust systems for autonomous vehicle navigation in challenging and unknown Terrain. J. Field. Robot. **23**(8), 509–526 (2006)
4. Mason, R., et al.: The Golum Group/ University of California at Los Angeles autonomous ground vehicle in the DARPA grand challenge. J. Field. Robot. **23**(8), 527–553 (2006)
5. Lakhotia, A., et al.: CajunBot: architecture and algorithms. J. Field. Robot. **23**(8), 555–578 (2006)
6. Travis, W., et al.: SciAutonics-Auburn Engineering's low-cost high-speed ATV for the 2005 DARPA grand challenge. J. Field. Robot. **23**(8), 579–597 (2006)
7. Miller, I., et al.: Cornell University's 2005 DARPA grand challenge entry. J. Field. Robot. **23**(8), 625–652 (2006)
8. Crane, C., et al.: Team CIMAR's NaviGator: an unmanned ground vehicle for the 2005 DARPA grand challenge. J. Field. Robot. **23**(8), 599–623 (2006)
9. Touchton, R., Galluzzo, T., Kent, D., Crane, C.: Perception and planning architecture for autonomous ground vehicles, IEEE Comput. **39**(12), 40–47 (2006)

10. Coppin, B.: Artificial Intelligence Illuminated, pp.108–109. Jones and Bartlett Publishers, Sudbury (2004). ISBN 0-7637-3230-3
11. Touchton, R., et. al.: Planning and modeling extensions to the joint architecture for unmanned systems (JAUS) for application to unmanned ground vehicles. In: SPIE Defense and Security Symposium, Orlando, March 2005
12. Solanki, S.: Development of sensor component for Terrain evaluation and obstacle detection for an unmanned off-road autonomous vehicle. Ph.D. dissertation, University of Florida (2007)
13. Nanning, Q., Cheng, H.: Springbot: a prototype autonomous vehicle and its algorithms for lane detection. IEEE Trans. Intell. Transport. Syst. **5**(4), 300–308 (2004)
14. Odagiri, K., Kobayashi, K., Watanabe, K., Umino, H., Nakaho, N.: Development of active contour extraction for autonomous vehicle lane detection at outdoor environment. SAE J. Automot. Eng. (2001)
15. Matthews, X., John, H.: Numerical Methods for Computer Science, Engineering and Math, pp. 224–252. Englewood Cliffs, Prentice-Hall, New Jersey (1987)
16. Lee, J., Crane, C., Kim, S., Kim, J.: Road following in an unstructured desert environment using monocular color vision as applied to the DARPA grand challenge. In: Proceedings of the International Conference on Control, Automation, and Systems (ICCAS 2005), Gyeong Gi, Korea, June 2005
17. Gonzalez, R.C., Woods, R.E.: Digital Image Processing. Prentice Hall, NJ (2002)
18. Viola, P., Jones M.J. Rapid object detection using a boosted cascade of simple features. In: Proceedings IEEE Conference on Computer Vision and Pattern Recognition, USA (2001)
19. Touchton, R.: An adaptive planning framework for situation assessment and decision-making on an autonomous ground vehicle. Ph.D. dissertation, University of Florida (2006)
20. Mayne, D.Q., Rawlings, J.B., Rao, C.V., Scokaert, P.O.M.: Constrained model predictive control: Stability and optimality. Automatica **36**(6), 789–814 (2000)
21. Scokaert, P.O.M, Mayne, D.Q., Rawlings, J.B.: Suboptimal model predictive control (feasibility implies stability). IEEE Trans. Auto. Cont. **44**(3), 648–654 (1999)
22. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Sys. Sci. Cyber. **SSC4**(2), 100–107 (1968)

Part III
Navigation

The *V*ictor*T*ango Architecture for Autonomous Navigation in the DARPA Urban Challenge

Patrick Currier, Jesse Hurdus, Andrew Bacha, Ruel Faruque, Stephen Cacciola, Cheryl Bauman, Peter King, Chris Terwelp, Charles Reinholtz, Alfred Wicks and Dennis Hong

Abstract To solve the autonomous urban driving problem presented by the 2007 DARPA urban challenge, team *V*ictor*T*ango developed a tri-level architecture with a deliberative-reactive-deliberative structure. The *V*ictor*T*ango architecture emphasizes a robust, reusable, and modular design, using hardware-independent virtual actuators and virtual sensors. Details of the urban challenge implementation are provided, highlighting the functionalities and interactions of different modules within the architecture. Situational examples from the urban challenge problem illustrate the performance of the architecture in real-world situations, and communications latencies with their effect on decision making are analyzed. Finally, recommendations for future refinement of the architecture are presented. The *V*ictor*T*ango architecture ultimately proved capable of completing the urban challenge problem. Furthermore, the modularity, hardware independence, and robustness of the architecture have enabled it to be applied to other platforms and other problems in the unmanned systems field.

P. Currier (✉) · A. Wicks · D. Hong
Virginia Polytechnic Institute and State University, Blacksburg, VA, USA
e-mail: pcurrier@vt.edu

D. Hong
e-mail: dhong@vt.edu

J. Hurdus · A. Bacha · R. Faruque · S. Cacciola · C. Bauman · P. King · C. Terwelp
LLC, TORC Technologies, Blacksburg, VA, USA
e-mail: info@torctech.com

C. Reinholtz
Embry-Riddle Aeronautical University, Daytona Beach, FL, USA
e-mail: reinholc@erau.edu

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_5, © Springer-Verlag London Limited 2012

1 Introduction and Overview

The 2007 DARPA Urban Challenge (DUC) was a landmark technical problem that pushed the envelope of mobile robotics technology. Problems such as robot cognition, sensor fusion, behavior coordination, and path planning were brought out of the lab and into a realistic field environment [5, 23]. Virginia Tech and TORC Technologies, LLC, formed Team *V*ictor*T*ango and developed a novel software architecture that was capable of integrating available hardware into a system capable of meeting the requirements of the urban challenge while being flexible enough to enable its use in future applications.

1.1 Chapter Overview

The DUC, as well as the preceding grand challenges, can be conceived of largely as a system integration problem. A fully-autonomous vehicle is a complex system that is required to sense its environment, process that sensor information, apply domain-specific knowledge (in this case the rules of the road) to select an appropriate action, and then execute the decision with the physical vehicle. The short time scale of the DUC (1 year start-to-finish) made development of problem-specific sensing and computing technologies nearly impossible. This limitation, combined with the ambitious objectives forced teams to approach the DUC by developing a system that integrated largely commercial off-the-shelf (COTS) sensing and computing hardware with a software suite capable of performing the required cognitive and executive tasks. The key to integrating these many parts together, and thus the key to solving the DUC, was the underlying software architecture.

Completing such sophisticated tasks in a dynamic, partially-observable, and unpredictable environment requires a flexible system architecture that balances both planning and reactivity. To accomplish this, team *V*ictor*T*ango developed a tri-level, hybrid deliberative/reactive architecture. This architecture takes advantage of optimal, search-based, planning techniques as well as reactive, behavior-based methods. It follows a deliberative-reactive-deliberative progression that sandwiches a behavioral component between a high-level mission planner and a low-level motion planner. World modeling is handled in parallel, providing sensor-independent perception messages to the control modules via virtual sensors. Hardware independence is then maintained through the use of virtual actuators. All software agents operate asynchronously and communicate according to the JAUS protocol, an emerging industry standard. As a result, the *V*ictor*T*ango software architecture is very modular in design, making it well-suited for rapid deployment by a team of developers in a situation such as the DUC that requires an immense amount of testing and design cycles in a very small time window. This is further aided by the inherent portability and scalability allowing easy adaptation for

simulation or hardware deployment. Finally, health monitoring of both hardware and software is continuously performed at different levels, allowing for various error recovery techniques that increase the robustness and reliability of the overall implementation.

In this chapter, the unique problem posed by the DUC and the hardware and sensing platform used by team *V*ictor*T*ango is presented. A brief history of autonomous mobile robot software architectures illustrates the progression of the field, and provides a context for discussing the improvements made with the *V*ictor*T*ango architecture. Next, a detailed discussion of the DUC implementation is given, focusing on the distribution of responsibilities and the balance between deliberative and behavior-based approaches. Finally, the flow of information and the chain of decision-making are presented in several different examples and aspects of the architecture's performance are analyzed. Finally, conclusions about the architecture are drawn and ideas for future work are presented.

1.2 Problem Definition

Although the problem presented by the DUC has been discussed previously in this book, it is important to revisit several of these aspects as they are relevant for the discussion of the architecture. The DUC required an autonomous ground vehicle to navigate an ordered list of checkpoints in a road network. The road network is supplied as a-priori information in the form of a route network definition file (RNDF) that lists each drivable road as well as regions called zones, used to represent parking lots or other unstructured environments. The RNDF describes each lane by a series of waypoints with certain waypoints flagged as entrances and exits that connect each lane to the road network. A series of mission data files (MDF) are provided that specify checkpoints which must be visited in a specified order. To achieve the MDF checkpoints as fast as possible, the vehicle must choose roads considering road speed limits, possible road blockages, and traffic conditions. The vehicle must obey California state driving rules while driving safely and defensively, avoiding both static obstacles and moving traffic at speeds of up to 30 mph. Finally, the autonomous vehicle must be able to complete the 60 mile competition in less than 6 h.

Several important observations can be made from an analysis of the basic requirements. First and foremost among these is that the basic environment is both known and controlled. The vehicles are only required to operate on pre-defined and clearly-distinguishable roads and the only mobile object encountered will be other vehicles (pedestrians were specifically excluded). Additional assumptions such as the RNDF waypoints correctly defining the road or that the behavior of other vehicles will conform to driving laws may be made, but may not be true in all cases. The vehicles are not required to proceed to unknown destinations or operate across unstructured terrain. Additionally all actions can be governed by a common set of regulations: the California state driving regulations as augmented

Fig. 1 Odin undergoing testing at Virginia Tech (*left*) and Front seat view of Odin (*right*)

by DARPA. A distinction must be made, however, between the highly-structured road driving and the unstructured zone environment as these require very different behaviors. The vehicles must also be able to adapt to changes in the RNDF caused by both planned and unplanned blockages. Additionally, there are no second chances in the DUC, so the vehicles must be able to handle and correct for errors that may occur without human intervention. These observations are critical for simplifying the problem to a manageable level and drove the design and implementation of the software architecture.

1.3 Hardware Overview

The base platform is a 2005 Ford Escape hybrid converted by the team to meet the DARPA requirement of an autonomous full-size commercial vehicle platform. The escape, dubbed Odin, is outfitted with a highly-integrated custom drive-by-wire system and is shown in Fig. 1. The rear cargo area holds the main computing system, sensor communication modules, and power distribution. Power for the autonomous systems is drawn from the hybrid battery pack and an uninterruptable power supply (UPS) is used for back-up power.

The computing system consists of two main systems mounted in the rear and a network of distributed embedded computing systems. The main systems were state-of-the-art in 2007 and are Hewlett–Packard servers each equipped with dual, quad-core Intel Xeon processors and 4–8 GB of RAM. One system runs Microsoft Windows XP as an operating system and the other system runs the Fedora Core 6 distribution of GNU/Linux. Further processing capability was provided by two proprietary computing units purchased as part of the IBEO sensor system and a National Instruments CompactRIO system that controlled low-level drive-by-wire functions. Primary communication between computing nodes is accomplished via a switch Gigabit Ethernet network. Separate communication modules convert data from serial-based sensors into Ethernet packets for transmission to the computing

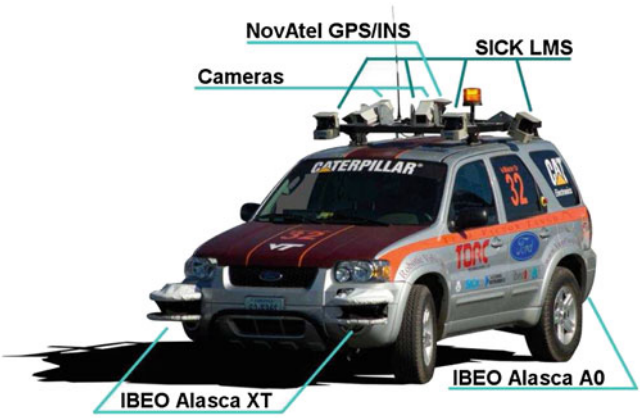


Fig. 2 Odin's sensor layout with sensors labeled

nodes. The distributed nature of the computing system was a driving factor in the design of the architecture.

Odin's exteroceptive sensor suite (Fig. 2) covers the full 360° azimuthal area around Odin. Primary obstacle detection is provided by three IBEO LIght Detection and Ranging (LIDAR) units. Two IBEO XT ALASCA LIDARs are mounted at the front corners of Odin, covering a combined 260° horizontal field of view, as shown in Fig. 3. These two sensors operate as a coordinated pair through IBEO's Fusion system. The third IBEO unit, an ALASCA A0, is mounted on the rear bumper and covers a 150° horizontal field of view. All three sensors contain four scan planes that span a maximum of 3.2 vertical degrees. The IBEO's are capable of processing multiple returns for each laser pulse, effectively allowing them to see through rain and light dust clouds.

Four SICK LMS 291 single-plane LIDAR units used to supplement the IBEOs are mounted on the roof rack. Two SICKs point forward and down at angles of 8 and 10° below horizontal, resulting in an approximate range on level ground of 12 and 10 m, respectively. These units were used to supplement the IBEOs' object detection capabilities by detecting small objects, ditches, and curbs. The other two SICKs are mounted on each side of the vehicle, and are used to monitor Odin's blind spots during lane changes.

Additional sensors include two IEEE-1394 color cameras mounted on the roof rack facing forward, covering a 90° horizontal field of view. These cameras are capable of transmitting raw 1024 by 768 pixel images at up to 15 frames/s, and are used to detect roads and assist in object classification. A NovAtel Propak LB-plus GPS/INS unit is used to provide a COTS inertially filtered global positioning solution. Coupled with an OmniSTAR HP differential correction service subscription, the unit has a rated CEP of 10 cm. Further details about the specific hardware implementation can be found in [8, 9, 24].

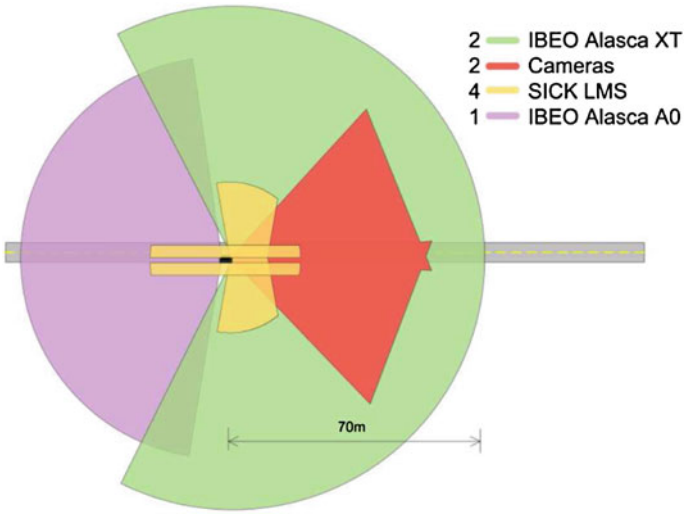


Fig. 3 Coverage provided by the sensors mounted on Odin (the *black rectangle* in the *center* is Odin)

1.4 Review of Software Architectures

The origins of mobile robot control are rooted in the hierarchical paradigm, as defined by Murphy in [18]. The hierarchical paradigm relies on the ability pre-plan the robot's actions over a fixed time interval. The emphasis on planning in this approach pushed the development of more effective search algorithms, such as Dijkstra and A*. These algorithms allow an agent to find an optimal "sequence of actions that achieves its goals, when no single action will do" [20]. While improved search and planning algorithms helped to reduce the time needed to plan, it was not enough to overcome the growing complexity needed to complete even simple tasks, such as navigating a cluttered room. As the number of objects being represented in the world increases and the number of possible actions broadens, the search space grows exponentially. Subsequently, the planning component became a bottleneck.

In response to these shortcomings, the reactive paradigm incorporates important ideas from ethology, the study of animal and insect behavior. The main feature of the reactive paradigm is that all actions are accomplished through the use of distinct behaviors. The overall architecture is built-up as a set of concurrently running behaviors. The result is that a robot's overall behavior emerges from the combination of behaviors operating at any given time [6]. Essentially, planning is discarded in favor of simpler, more efficient, and reactive behaviors. The result is a parallel, vertical decomposition of behaviors that is much faster in operation, requires less computational power, and is more robust to failure. However, it is also much more imprecise. It is difficult to predetermine exactly what discrete

behaviors are needed to combine and produce a desired rich emergent behavior. Even when the correct-base behaviors are determined, exactly how they are combined has a huge effect on what emergent behavior results. This is known as the action selection problem. Furthermore, without any sort of planning component, purely reactive architectures eliminated any form of memory or reasoning about the global state of the robot. Not only could optimal trajectories not be calculated, but functionality such as map making and performance monitoring were lost. In total, it became very difficult to design a set of behaviors that would be wholly sufficient for completing more complex, multi-objective tasks.

Due to the shortcomings of the reactive paradigm, there emerged a need to find a way to re-incorporate some of the high-level cognitive abilities of the hierarchical paradigm. Techniques were needed for sequencing or assembling behaviors such that a series of sub-goals can be achieved as necessary for more complex problems. The result is the bi-level hybrid deliberative/reactive paradigm that presents the idea of combining reactive behaviors with top-down, hierarchical planners used for more deliberative functions. These deliberative components are designed to run asynchronously, providing a set of intermediary goals for use as guidance to a lower-level reactive system. These intermediary goals are intended to be sufficient for preventing the reactive system from making too many poor decisions as well as provide the ability to recover from being 'trapped'.

A loose analogy of hybrid architectures can be made to the way the human brain works. Large segments of the human brain can be considered to be associated with certain specific abilities, from sensor processing in the visual cortex to motor control in the brain stem. The same role-based segmentation exists in hybrid architectures as individual software modules. These software modules are free to use whichever paradigm is most appropriate for their individual task, whether it be search-based or behavior-based. This allows for a deeper abstraction of responsibilities within the overall control architecture. Such an approach also lends itself well to object-oriented programing and the portability of specialized modules to new domains. The question then becomes how to organize these software modules and determine what information should be passed between them. How does the overall behavior emerge? When properly implemented, a hybrid approach balances reactivity to unforeseen events with the execution of planned actions, similar to what is seen in biological intelligence. Figure 4 illustrates this problem and analogy to the human brain.

The question remains as to where to draw the line between planning components and reactive components. The early success of well-known reactive, or behavior-based architectures such as subsumption architecture [6] or potential fields [13] in the mobile robot navigation domain were due to their speed, elegance, and simplicity. Reactive architectures subsequently became accepted as the "best" way of doing low-level motion control [18]. This led to most hybrid architectures having a single division between deliberative and reactive layers. Cognitive modules responsible for mission planning, map making, performance monitoring, and behavioral sequencing are done on the higher, deliberative layer, while selected behaviors run on the lower, reactive layer, interacting directly with

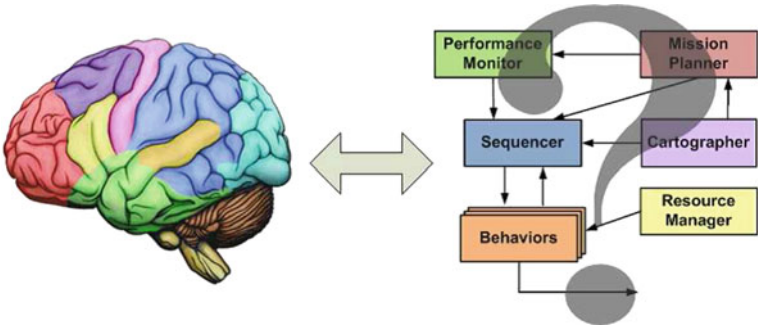


Fig. 4 Human brain/hybrid architecture analogy

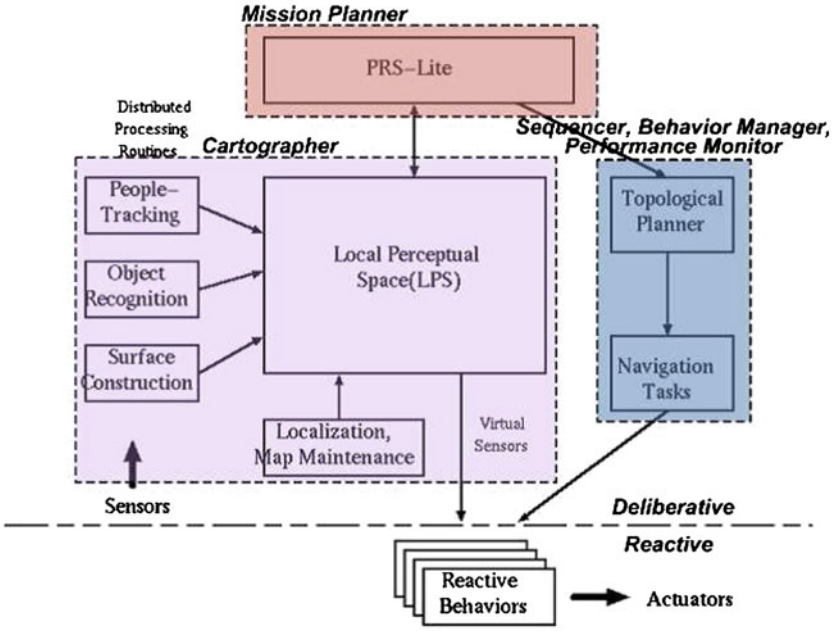


Fig. 5 The Saphira architecture (bi-level)

the robot's actuators. Many well-known hybrid control architectures were developed with this bi-level approach, such as the autonomous robot architecture (AuRA) [4], sensor fusion effects (SFX) [17], 3T [10], task control architecture (TCA) [21], and Saphira [14]. The Saphira architecture, developed at the Stanford Research Institute (SRI) for use on the Pioneer robot platform is a good example of the bi-level hybrid approach and is illustrated in Fig. 5.

The deliberative layer is needed to bring robotic applications into more cognitively-challenging domains beyond just navigation. The reactive layer is then

responsible for more time critical control problems such as low-level obstacle avoidance, allowing the robot to react to unforeseen events and surprises. Due to the nature of the reactive paradigm, however, there is no guarantee of optimality beyond the intermediary goals set by a deliberative planner. Even if the robot goes in the right direction, it could easily spend extra time getting into and out of box canyons or other situations where behavior-based navigation algorithms have problems. Fortunately, with the rapid growth of computing technology, there has been a re-emergence of deliberative methods for low-level Motion Planning [3, 22]. Path-planning algorithms that once took minutes to run can now run multiple times in a second. Because of this, utilizing deliberative planning techniques for low-level motion control is much more realistic. A new type of hybrid deliberative-reactive architecture is needed to take advantage of these methods, while maintaining the benefits of behavior-based systems.

2 *V*ictor*T*ango's Tri-Level Architecture

To address the needs presented previously, the team developed the *V*ictor*T*ango architecture, a tri-level hybrid deliberative/reactive software architecture. This architecture contains perceptive, deliberative, and reactive software components that will be discussed in detail along with the distribution of responsibilities within the architecture. Additionally, the system-level health monitor increases reliability and a communications backbone that enables the portability and scalability of the architecture.

2.1 Architecture Overview

Traditionally, hybrid architectures utilize a bi-level layout with only one division between deliberative and behavior-based components. This results in robust low-level motion control, but lacks the optimality and repeatability needed for certain applications, such as the DARPA urban challenge. Obstacle avoidance and lane maintenance on a 4,000 pound commercial vehicle moving at 30 mph requires a level of predictability and optimality in Motion Planning that is difficult to achieve with behavior-based motion control.

To address this problem, *V*ictor*T*ango has incorporated technological advances in model-predictive Motion Planning into a *tri-level* hybrid architecture. In this tri-level approach, there still exists a deliberative layer at the highest level, responsible for such tasks as mission planning and map making. These modules then provide intermediary goals to a reactive, behavioral component that is then sandwiched by yet another low-level deliberative layer. The end result is a *deliberative-reactive-deliberative* progression. The low-level deliberative layer is strictly responsible for Motion Planning and sends the final output to the Vehicle

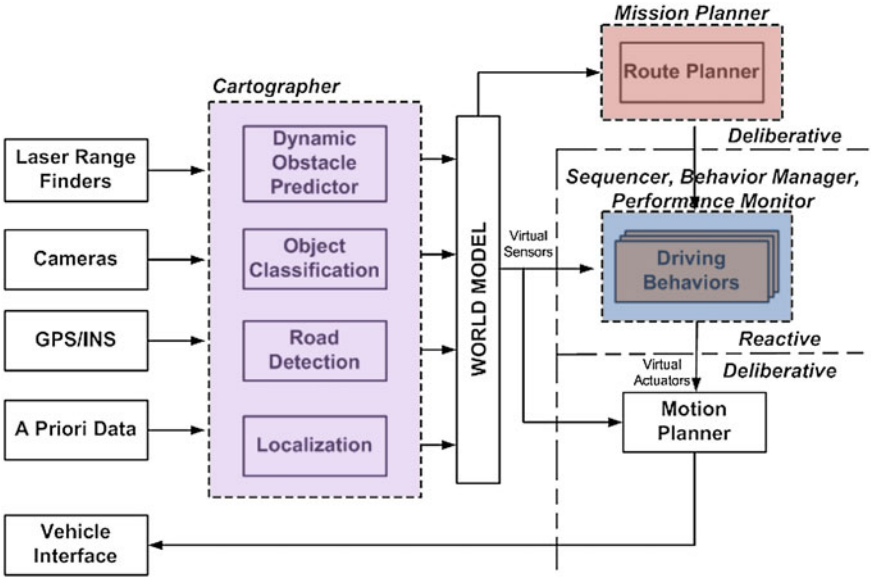


Fig. 6 The *Victor*Tango architecture (tri-level)

Interface. It is only concerned with events on a short time horizon and utilizes optimal planning techniques to find the best series of actuator outputs. Unlike in the Hierarchical paradigm, where a plan is made and then the robot attempts to follow that plan until a problem occurs, the low-level path planner continually replans at a high rate.

*Victor*Tango's tri-level software architecture is shown in Fig. 6. In this architecture, a deliberative Route Planner is run solely on demand, when a new mission is loaded or if a road-block is encountered. This planned "route" is then provided to Driving Behaviors, a reactive, behavior-based module responsible for maintaining situational awareness while balancing the rules of the road with the goals set forth by the Route Planner. Finally, high-level motion commands are sent to the motion planner, which uses deliberative search methods to find the optimal path through the immediate environment. Similar to Saphira, all sensor data is routed through a set of perception modules responsible for locating, identifying, and tracking specific percepts such as the road, static obstacles, and dynamic obstacles. These sensor-independent perception messages compose the global world model and are provided to the behaviors and planning components as virtual sensors. Perception can therefore be seen as a parallel process in which the same representation of the world is used by all three decision-making layers despite the lack of a traditional monolithic "world model" in the architecture.

Since a deliberative approach is being used for low-level obstacle avoidance, dynamic stability, and lane maintenance, the scope and responsibility of the behavior-based software agent shifts upward. The need now exists for a behavioral

component capable of complex decision-making to complete a variety of abstract, multi-faceted, temporal problems. Specifically, this behavioral module is responsible for providing contextual intelligence, monitoring the situation at hand, and taking into account the changing goals and sub-goals of the vehicle. Through a method of behavior coordination, acceptable emergent behavior given the current situation must be produced through the issuing of high-level, hardware-independent motion commands, or virtual actuators.

2.2 Perception

Odin's perception is handled by a collection of software modules intended to interpret and translate specific sensor information into a small set of virtual sensors, or sensor-independent perception messages, for the planning modules. These virtual sensors describing the location of lanes, drivable area, static obstacles, and dynamic obstacles, and the vehicle's pose compose the world model that the Planning software acts upon. By defining virtual sensors that are not dependent on specific hardware, the addition of another sensor or sensor modality affects only the perception software, improving the overall performance of Odin while enhancing the reusability of the decision-making software for future projects.

While perception is an extremely important operation of an autonomous vehicle, the real innovation in the *V*ictor*T*ango architecture is contained in the decision-making aspects. The most important concept in the perception part of the architecture is that of virtual sensors. This concept allows the decision making modules to be abstracted from the actual algorithms used to generate the relevant messages, allowing the processing algorithms to be modified as necessary to accommodate differing environments. As a result, an extensive discussion of the algorithms used to generate the virtual sensor messages will not be included in this work. Further discussion on these algorithms can be found in the team's paper in the journal of field robotics [5].

2.2.1 Decoupled Local and Global Frames

Odin's Localization module is responsible for reporting two decoupled position solutions: a GPS-based global position and an inertial local position. Used in primarily referencing the RNDF, the global position relies primarily on a COTS filtered GPS/INS solution. This solution is fused with measured wheel speeds and steering angle using an extended Kalman filter designed for Odin, based on standard practices [12]. Implemented to assist the GPS/INS solution during times of poor GPS signal or GPS outages, the filter also adds to the scalability of the module because the inherent nature of the filter allows for future measurements (expansion of perception capabilities) and redundant information (more robust through statistical weighting).

The second position solution is an entirely inertially-based local position. Starting from an arbitrary origin, Odin tracks its location through the local frame using wheel speeds, steering angle, and inertial measurements. The error associated with this frame is typically within 2.6% of the distance traveled, an error accumulation not significant within the sensor range. The goal of this frame is to provide a smooth and continuous position estimate for obstacle tracking and short-range planning that is devoid of the potential GPS jumps or errors. Such discontinuities, known as "GPS pop", can be disastrous to object tracking and prediction algorithms that compare sequential frames of data. An instantaneous local-to-global transformation calculated with the local solution allows locally detected objects or roadway information to be transformed into the global frame for comparison against the RNDF. For more information on the localization software, see [24].

2.2.2 Road Detection

The road detection software component determines the roads and zones that are available for navigation, providing its output in two messages: lane position and drivable area coverage. For each lane, the lane position message provides the position and width of the travel lane, as well as all possible branches (smooth paths to other lanes) at upcoming intersections. Drivable area coverage expresses road and zone data as a vehicle-centered, 2-D binary map; for example, in a zone, a parking lot island would appear as an undrivable area. Both outputs are expressed in the local frame. Road data is used in object classification for filtering out objects not close to a road, in the dynamic object Predictor for calculating likely trajectories of other traffic, and in Driving Behaviors and Motion Planning for planning. For more information on the road detection software, see [5].

2.2.3 Object Classification and Dynamic Object Predictor

Object classification is responsible for merging data from multiple sensors to detect and classify nearby objects. Objects are classified into two categories: static obstacles that do not move, and dynamic objects that are in motion or have the potential for motion; for the urban challenge, the only dynamic objects Odin was expected to encounter were other vehicles. Objects are provided to downstream software modules as a list of static objects (with profile points), and a list of dynamic objects (with a length, width, heading, speed, stopped time, and profile points). All locations are transformed from vehicle to local frame before transmission.

While static objects are passed directly to Motion Planning, the list of dynamic objects (vehicles) is passed to a dynamic object predictor, which is responsible for classifying each vehicle in a lane and determining likely paths for each vehicle, incorporating lane position information from road detection. A vehicle's path is

determined by examining its motion history and relevant lane position data for the segment it is on; if there is no available lane data, as is the case inside a zone, the path is projected based on the vehicle's current motion. Once the set of paths is calculated, it is appended to the vehicle's data as a series of discrete positions, each with an approximate arrival time. The final dynamic object list is used by Driving Behaviors and Motion Planning for traffic behavior and obstacle avoidance. For more information on the object classification software, see [8].

2.2.4 Perception Focus

The perception focus feature in the architecture exists due to limitations in the hardware. The IBEO LIDAR units are only capable of outputting 64 objects, a limit that is exceeded in many real environments. To address this limitation, it was necessary to limit the object output of the IBEO system to a zone centered along the longitudinal axis of the vehicle. In certain situations, such as merging or exiting across an oncoming traffic lane, it is beneficial to have perception extend its sensory range in one or more directions, at the expense of another direction. To enable this, the road detection, object classification, and dynamic object predictor modules can receive a "set perception focus" message instructing them to extend focus left, right, rear, or forward. This feature allows the autonomous vehicle to effectively "look both ways" while merging. This method bypasses an inherent hardware limitation without sacrificing capability and may also prove useful in future implementations where it is necessary to limit the focus of computationally-intensive algorithms.

2.3 Decision-Making

*Victor*Tango's decision-making software is broken into four main parts: a deliberative Route Planner that establishes high-level mission goals, a reactive Driving Behaviors component to follow traffic laws by spooling short-term goals with constraints, a deliberative motion planner that establishes a safe path through the local area to achieve the short-term goals, and a Vehicle Interface that commands the actuators to execute the plan.

2.3.1 Deliberative Route Planner

The Route Planner software module is responsible for choosing which roads are traveled to complete a mission. The only software component on Odin that runs on demand rather than periodically, the Route Planner generates a route whenever a new MDF is loaded or if the current road segment is completely blocked, requiring a new route. The output of the Route Planner is a list of waypoints that are exits,

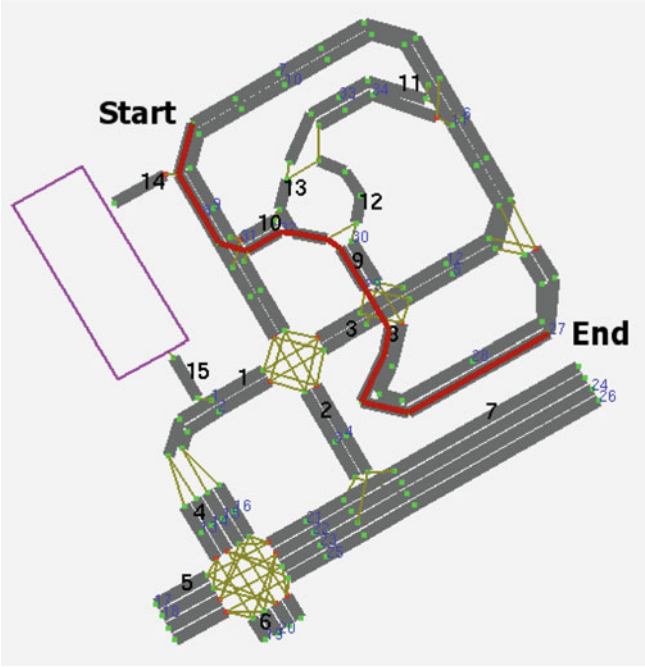


Fig. 7 The Route Planner selects the road exits and entrances used to complete a mission, shown as highlighted route

entrances, or checkpoints. Any waypoints along the road lane are omitted, as the Route Planner does not choose the lane of travel if there are multiple lanes in a road. Checkpoints, however, must still be specified to ensure the vehicle is in the proper lane when passing a checkpoint. An example Route Planner output is shown in Fig. 7.

The implementation of the Route Planner is based on an A* graph search. Only a limited subset of world information is considered by the Route Planner. The Route Planner uses the road information contained in the RNDF file, Speed Limits defined in the MDF file, and road blockage reports from the Driving Behaviors software module rather than direct obstacle information. The exit and entrance pairs of the road network are used to form a graph and an A* search is used to minimize travel time from the current vehicle location to the mission checkpoints. Travel time is estimated by applying the MDF Speed Limits to the distances between lane waypoints. To account for time at intersections and reduce traffic interactions, time penalties can also be applied to certain events. For the urban challenge, the time penalties used were: 30 s for any exit, 180 s for a U-turn, and 400 s for entering a zone. Another final event parameter was the distance needed for a lane change, which was set to 20 m. For example, if a road had two forward travel lanes, and Odin entered the right lane, the Route Planner would only connect exits in the left lane that were at least 20 m down the road. The completed route is

then output to the Driving Behaviors module and the Route Planner remains idle until a new MDF is loaded or a blockage is encountered.

2.3.2 Reactive Driving Behaviors

Driving Behaviors is responsible for maintaining situational awareness, obeying the rules of the road, and interacting with other traffic while achieving the mission. This includes choosing which lane to drive in, deciding when lane changes are necessary, determining and respecting the progression order at intersections, handling blocked roads, and reacting to other unexpected traffic situations. To accomplish this high-level task, Driving Behaviors spools short-term goals and motion guidelines to the motion planner via a set of virtual actuators that make up a behavior profile.

Implementation—as mentioned in the architecture overview, the role of the reactive, behavior-based layer has shifted upward for bridging the gap between high-level mission planning and low-level navigation. Specifically, the Driving Behaviors module provides two important aspects of embodied A.I., contextual intelligence and emergent behavior. Contextual intelligence provides the robot with a mechanism of understanding the current situation. This situation is dependent on both the current goals of the robot, as defined by the Route Planner, as well as the current environment, as defined by the virtual sensors. Such insight is important for performance monitoring and self awareness along with the ability to balance multiple goals and sub-goals. Emergent behavior is a very important trait of biological intelligence which we understand to be necessary for the success of living organisms in the real world. It allows for the emergence of complex behavior from the combination of simpler behaviors, which is important not only for individual intelligence, but cooperative intelligence within multi-agent systems as well.

In total, Driving Behaviors is tasked with selecting the most 'appropriate' actions, taking into account the mission plan, sensed traffic, sensed road blockages, and the rules of the road. The process of deducing the most appropriate action is known as the action selection problem and solutions to this problem are known as action selection mechanisms (ASM). Existing ASMs, according to [19], fit either in the arbitration or in command fusion class, as shown in Fig. 8.

Arbitration ASMs allow "one or a set of behaviors at a time to take control for a period of time until another set of behaviors is activated" [19]. Arbitration ASMs are therefore most concerned with determining what behaviors are appropriate given the current situation. Once this has been determined it is guaranteed that there will be no conflict in outputs between the running behaviors and so no method of combination or integration is needed. Command fusion ASMs, on the other hand, "allow multiple behaviors to contribute to the final control of the robot" [19]. Rather than being concerned with selecting appropriate behaviors, command fusion ASMs let all behaviors run concurrently, then rely on a fusion scheme to filter out insignificant behavioral outputs. Command fusion ASMs are

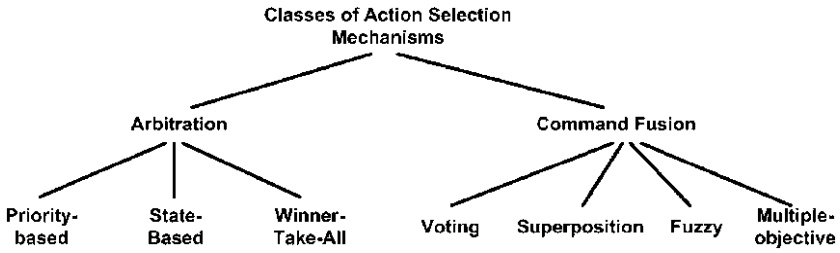


Fig. 8 Pirjanian's taxonomy of ASMs

therefore typically described as being flat. Since multiple behaviors can end up desiring the same control, these ASMs present novel methods of collaboration among behaviors. This cooperative approach, rather than competitive, can be extremely useful in situations with multiple, concurrent objectives. On the other hand, arbitration mechanisms are more efficient in their use of system resources. By selecting only one behavior from a group of competing behaviors, processing power and sensor focus can be wholly dedicated to one task.

The novel, behavior-based architecture developed by *V*ictor*T*ango for the Driving Behaviors component preserves the strengths of both arbitration and command fusion ASMs [11]. This is possible by placing an arbitration ASM in series with a command fusion ASM. The result, in essence, is the ability to select a subset of behaviors given the current situation. Then, if multiple behaviors competing for the same output are activated, they can still be cooperatively combined using a method of command fusion.

Arbitration Mechanism—a state-based, hierarchical, arbitration ASM is used for behavior selection. This method utilizes a hierarchical network of finite state automata (FSA), which can be referred to as a hierarchical state machine (HSM). Using a hierarchical approach to behavior decomposition is a common practice in ethology. It allows for the differentiation of behaviors according to their level of abstraction [15]. Within the behavioral HSM, each individual state machine refers to a single behavior. Behaviors are organized in the hierarchy with more abstract behaviors higher in the tree, and more physical behaviors lower in the tree. Each behavior, or node, in the tree is responsible for determining which of their sub-behaviors should be activated. This is determined by each behavior's internal state and is not limited to only one sub-behavior. All behaviors have implied relationships based on of their position within the hierarchy tree. Behaviors can have parent–child relationships, or sibling relations, but is important to note that these relationships do not necessarily imply importance or priority. While some arbitration ASMs use hierarchy to determine the relevance of a behavioral output [6], this approach uses hierarchy solely as an abstraction method for task decomposition. Simply put, the primary function of the hierarchical tree is to determine what behaviors to run. Using a hierarchy allows us to logically break down a complex task into smaller, more manageable pieces. A simplified form of the behavioral HSM used on Odin for the Urban Challenge can be seen in Fig. 9.

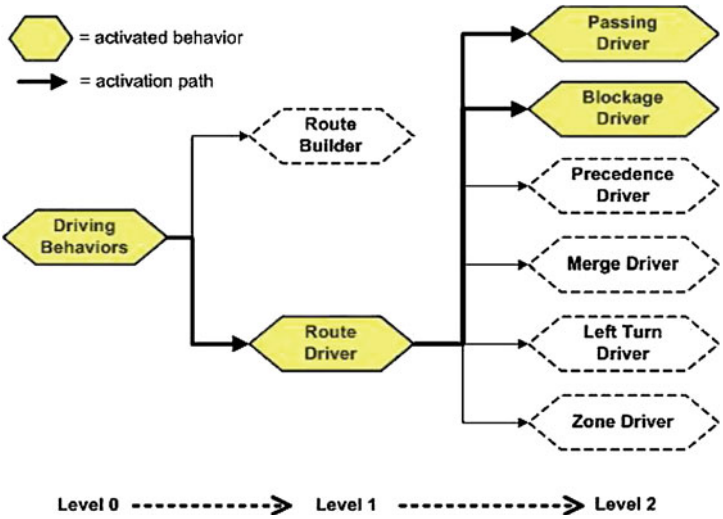


Fig. 9 Behavioral HSM used while driving in an urban environment

All behaviors are classified as either command and/or decision behaviors. A command behavior produces one or more VA commands, and a decision behavior results in the activation of a lower sub-behavior (i.e. a non-leaf node). Each behavior can be formally described as consisting of a set of controls states, $cs_i \in CS$. Each control state encodes a control policy, π_{va}, which is a function of the robot's internal state and its beliefs about the world (virtual sensor inputs). This policy, π_{va}, determines what action with respect to a specific VA to take when in control state cs_i. All behaviors have available with them the same list of virtual actuators $va_i \in VA$. Furthermore, each control state has hard-coded what sub-behaviors $sb_i \in SB$ to activate when in that state. Transitions between control states occur as a function of the robot's perceptual beliefs, in the form of virtual sensors, or built-in events, such as an internal timer. While each behavior may have a 'begin' and 'end' state corresponding to the start and completion of a specific task, a single behavior, or state machine, cannot terminate itself. The higher, calling behavior always specifies what sub-behaviors should be running. Should a sub-behavior complete its state sequence and have nothing to do, it will remain in an idle state and not compete for control of any VA.

Command behaviors in Fig. 9 are labeled as "drivers". Level 1 behaviors are primarily concerned with traversing the course solely on a priori information. These behaviors utilize the given RNDF and other a-priori information to execute as if no surprises in the world exist. Level 2 behaviors are then responsible for handling events that cannot be planned for, such as intersection traffic, disabled vehicles, and roadblocks.

Command Fusion Mechanism—with all the command behaviors defined as individuals, it is necessary to look at how these behaviors interact as a group. Since

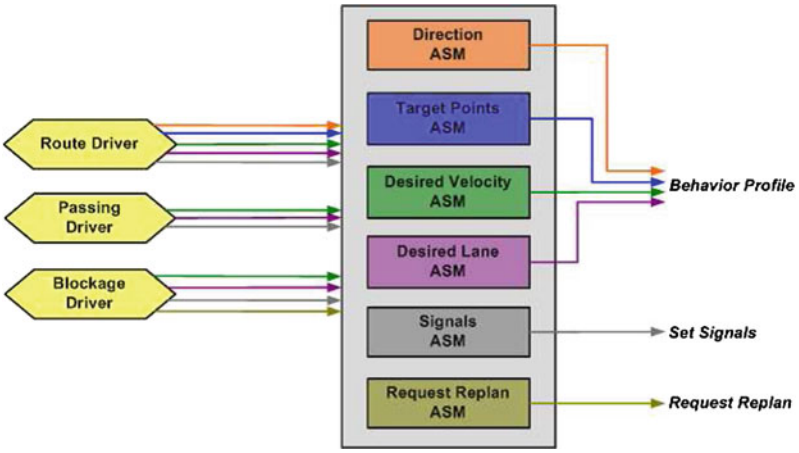


Fig. 10 Command Fusion ASMs used on Odin

control policies are classified by virtual actuator, the main interaction between behaviors occurs when two or more control policies are outputting to the same virtual actuator. For example, when π_{lane} of the Passing Driver disagrees with π_{lane} of the route driver, some form of resolution is needed. When this situation occurs, it is the responsibility of the command fusion mechanism to produce the final desired lane that will be bundled in the behavior profile.

Once the final set of behaviors has been selected, they are placed in a flat organization. A command fusion mechanism is then used for each virtual actuator to determine the final output. This command fusion mechanism can utilize many existing techniques [3, 6, 13] or implement a novel method. The choice of command fusion mechanism is dictated by the robot application and the specific virtual actuator. For example, virtual actuators that have only a finite set of possible commands do not lend themselves well to superposition based ASMs.

Figure 10 illustrates the command fusion structure used in the DARPA urban challenge. For simplicity, only the route driver, passing driver, and blockage driver are activated in this example. All of these behaviors have their own set of control policies, some of which overlap with each other. When this occurs the respective command fusion ASM resolves the conflict.

For the Urban Challenge, it was determined that the same fusion mechanism was appropriate for all virtual actuators. The chosen mechanism uses a modified voting-based policy. Due to the rule heavy nature of urban driving, and the presence of well-defined individual maneuvers like passing, a non-superposition-based ASM was needed. Continuing the example from above, should the route driver and the passing driver desire to be in different lanes, it is not acceptable to drive in between two lanes. Instead, one behavior should take priority and take complete control of that virtual actuator. To make this possible, each behavior has encoded in all of its control policies an urgency value variable. This value indicates the priority of that

behavior generated by its control policy. The command fusion mechanism then has the simple responsibility of selecting the control policy with the greatest urgency. When only one behavior has an active control policy with respect to a virtual actuator, this decision becomes trivial.

Hierarchy with Parallelism—*V*ictor*T*ango's approach to behavioral programing presents the idea of using hierarchy with parallelism, features that are typically mutually exclusive. It has been shown in [7] that the major bottleneck in developing behavioral intelligence is not selecting the best approach or architecture, but developing the correct version of this approach. While complexity is needed for multi-faceted problems, reducing complexity is important for making the robot designer's job simpler. Furthermore, it is not enough that a behavioral system be able to do a lot of things, it is equally important that they do all those things right, and at the right times.

In real-world applications with major repercussions for incorrect behavior, performance predictability can be paramount. The ability to hand code behaviors and ignore certain perceptual triggers at certain times is extremely useful. Hierarchy takes advantage of selective attention to make this hand-coding of behaviors possible and practical. Yet at the same time, complex combinations of behaviors are important for developing higher level intelligence. Combining hierarchy with parallelism in the method presented here provides important flexibility to the behavioral programer. Situations in need of predictability can be catered to, while other situations can still take advantage of complex, parallel, combination schemes. This approach balances quantity and complexity with design practicality.

Messaging—At its core, the behavior profile sent to Motion Planning comprises six target points, expressed in local coordinates. Each point contains the lane and lane branch (for intersections), and an optional stop flag and heading criteria. A desired maximum speed, travel lane, and direction (forward, reverse, do not care) are also included. Finally, the zone and safety area (intersection) flags can be turned on to enable different modes within Motion Planning. As the vehicle progresses, Driving Behaviors receives feedback from Motion Planning on the achievability of the currently commanded goals, as well as notification when a target point is achieved. Driving Behaviors also commands the turn signals and horn directly to the Vehicle Interface and specifies a perception focus to the perception modules.

2.3.3 Deliberative Motion Planning

The Motion Planning software module is the final layer of planning before the purely executive low-level vehicle control. Motion Planning controls the speed and path of Odin to stay in the specified lane, avoid obstacles, and obey limits set by higher level software components. Unlike in many previous architectures, the Motion Planning module in the *Victor*Tango architecture is based on a primarily deliberative model. Instead of a set of behaviors, it relies on map-based Trajectory Search algorithms to select an appropriate vehicle path. This approach is capable

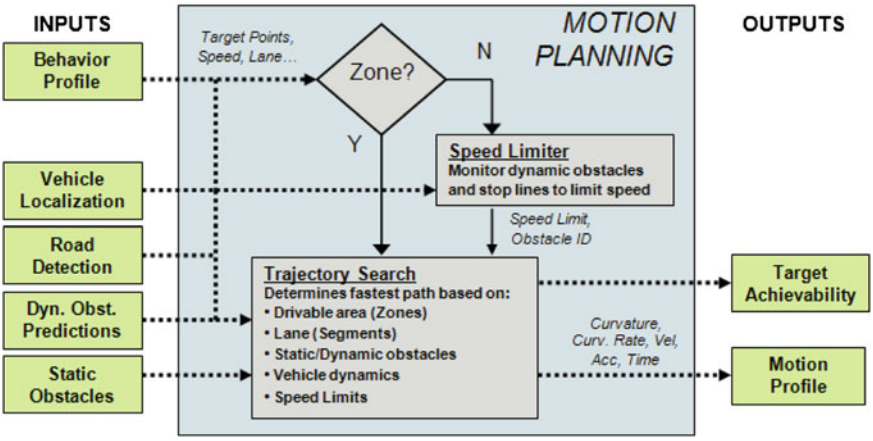


Fig. 11 Software flow diagram of Motion Planning component

of generating more optimal and predictable trajectories than a purely behavior-based approach.

Implementation—Motion Planning is based around a Trajectory Search component that maps local objects and then performs a deliberative search through the generalized cost map to find an optimal trajectory. One limitation of this approach is that, due to the generality of the implementation, it is not easy to encode the proper behaviors for dynamic situations, such as moving traffic. For example, when a slower moving vehicle is in the same lane Odin should continue to follow the same path as if the road were clear, but must slow down to react to traffic. To resolve this issue, a reactive element known as the Speed Limiter was incorporated into Motion Planning. The Speed Limiter monitors the road for traffic and stop lines and sends the Trajectory Search component a maximum speed and the ID of any vehicle that is effecting the maximum speed. The Trajectory Search will ignore the traffic specified by the Speed Limiter and plan a safe path using the lane data, static obstacles, and other dynamic obstacles. Trajectory Search still plans both a path and final speed, but the actual output is limited to the lower of the two speeds. The interactions between these components is illustrated in Fig. 11 which shows the inputs and interactions of Motion Planning.

The Trajectory Search component plans the future path of Odin by creating a local area cost map using road and obstacle data, and searching pre-computed trajectories to form a continuous path that achieves the behavior profiles set by Driving Behaviors. Trajectory Search uses a database of simulation results that form a coverage map of future vehicle occupancy for a given initial state and command. The search process consists of selecting trajectories using an A* search and evaluating for any collisions with obstacles or lane boundaries until the goal criteria is met, such as a distance down the lane. The search produces a smooth path by trying to minimize the travel time necessary. Obstacles are entered into the cost map with a high cost at the obstacle location and radiate decreasing cost.

This favors keeping a minimum distance from all obstacles, but allows tight maneuvering if all other options have been exhausted. For more information on this implementation, see [5].

The Trajectory Search is a deliberative component, but it should be noted that this component operates only in the local environment (within approximately 50 m of the vehicle) and does not consider mission level goals. This is an important distinction that enables a deliberative type Motion Planning search in a complex environment to be computationally feasible in real time. Additionally, the architecture allows for a very robust general-purpose motion planner to be developed without the need to encode specific domain knowledge at this level (the domain knowledge is contained in the behavior level); this division greatly enhances reusability of the base Motion Planning component.

By adapting a deliberative solver so that it can be run at a high rate, the Motion Planning in the *V*ictor*T*ango architecture gains the primary advantages of a deliberative system: increased optimality and increased predictability. Both of these characteristics take on a much greater importance as the speed capabilities of mobile robots continue to increase. Additionally, the deliberative style of Motion Planning tends to be much more robust to 'trap' situations due to its ability to consider future actions.

The Speed Limiter is largely reactive and starts by identifying which dynamic obstacles could enter Odin's path. The dynamic obstacle predictor assigns lanes to each obstacle, which the Speed Limiter compares to the future path of Odin. To ensure safety, any part of a vehicle within a lane is sufficient to be assigned to the lane. The Speed Limiter calculates the minimum clearance to pass another vehicle within the same lane, allowing Odin to pass parked cars on the side of a lane, slowing down if there is a tight clearance, but not attempt to pass a slower vehicle that is mostly within a single lane.

Architecturally, Speed Limiter operates in a manner much more similar to the algorithms used in Driving Behaviors than to the rest of Motion Planning. It is conceivable that the Speed Limiter could have been included in the Driving Behaviors component. However, since its output was very specific to Motion Planning and since it would have required an expansion of the Driving Behaviors messaging, it was decided to incorporate it as a part of the Motion Planning component.

Messaging—Motion Planning receives all perception messages, but operates entirely in the local coordinate frame and thus does not use the global position. Motion Planning receives commands from Driving Behaviors in the form of a behavior profile message. The behavior profile sets maximum speeds, specifies the lane of travel, and can also specify a desired heading for situations like parking where Odin should be aligned with the parking spot. Since Motion Planning receives data from many different sources at different rates, the Motion Planning software runs with a faster cycle time than most other components, typically 80 ms (see Table 2 for comparison). This faster update rate allows Motion Planning to incorporate the latest data as it continuously replans.

Table 1 Contents of motion profile message

Field no.	Name
1	Number of commands, X
1 + 5x + 1	Desired velocity (m/s)
1 + 5x + 2	Acceleration limit (m/s^2)
1 + 5x + 3	Path curvature (1/m)
1 + 5x + 4	Rate of curvature change (1/m/s)
1 + 5x + 5	Time duration (s)

Motion Planning produces motion commands called motion profiles that are sent to the Vehicle Interface and also can send messages to Driving Behaviors if the desired lane is not safely achievable. The motion profile provides a platform-independent message that specifies the path of travel, leaving the Vehicle Interface to translate into vehicle actuation. In practice both the motion planner and the Vehicle Interface will have knowledge of the vehicle platform dynamics, but a platform-independent message allows reuse of the message in other systems, with the Vehicle Interface incorporating additional parameters such as understeer that do not need to be planned for in the Motion Planning level. The motion profile message sets both the path and the speed, as shown in Table 1.

The motion profile message contains a series of commands where each command has a time duration. The Vehicle Interface executes the first command until the time duration expires, and then begins to execute the next command. This process repeats until a new motion profile message is received, or until there are no more commands to process. When a new motion profile message is received, the previous message is completely discarded and the new profile is executed immediately. The incorporation of an execution time functions as a safety mechanism. If the Motion Planning component unexpectedly exceeds its cycle time, this feature allows the vehicle to continue to travel along the last known safe path until Motion Planning can recover. If the time duration of the last command in a motion profile expires, the Vehicle Interface enters a safety mode and brings Odin to a stop.

While executing a command, the Vehicle Interface attempts to attain the desired velocity as aggressively as possible without violating the acceleration limit. The velocity is held at the desired velocity until the time duration expires, although the vehicle may not reach the desired velocity if the time duration or acceleration is set too small. The curvature works in a similar manner, where the curvature is held at the desired set point until the time duration expires, limited by the specified rate of change.

2.3.4 Vehicle Interface

The *V*ictor*T*ango architecture is designed such that the vehicle is a separate unit from the perception and planning modules. The design of the architecture provides

several advantages by separating the hardware interface and low-level control systems from the planning modules. The independence of the low-level control systems removes the need for the Motion Planning algorithm to have a detailed internal model of the vehicle dynamics. A basic linear model of a generalized Ackerman-steered vehicle can be defined using only a few basic performance parameters for use in Motion Planning. Generalized motion parameters such as path curvature, speeds, and rates of change can be used to define virtual actuators that are used to command the Vehicle Interface. The Vehicle Interface can then be tuned to transform the virtual actuators into actual actuator commands that correct for the true nonlinear dynamics of the vehicle.

The use of an independent Vehicle Interface commanded by virtual actuators also enhances the future reusability of the planning algorithms. The planning modules could be transferred to another similar autonomous platform by changing only a few basic parameters describing maximum vehicle steering and speed capabilities. This has been successfully demonstrated with Odin's software, which has been reused on other vehicles for other projects after the urban challenge.

2.4 Health Monitor

For Odin to operate completely autonomously for long periods with absolutely no human interaction, it becomes critical to implement a strategy for fault detection and health monitoring. Even seemingly simple errors such as a brief communications timeout can result in a catastrophic vehicle crash if the proper precautions are not taken.

In the *V*ictor*T*ango architecture health monitoring is handled at two levels. First, all software was designed with safe failure modes in mind; for example, if the motion planner is receiving no speed feedback, it defaults to commanding a zero speed. Secondly, each software module is responsible for detecting errors relevant to itself and reporting them to a central health monitor. These errors may range from an RNDF parsing error to a cycle time error to an unsatisfactory update rate in a message being received from another module and are reported with an error code, a human-readable string, and a criticality (warning, serious error, or catastrophe). Based on the cumulative set of errors, the health monitor decides whether the vehicle can proceed given its current state, and if necessary, issues a software pause command to the Vehicle Interface (zeroing the speed while retaining steering control with the emergency flashers on). Once the fault has been cleared, Odin resumes normal operation.

In certain driving situations, such as crossing a moving traffic lane when entering a main road, it is actually more dangerous to stop immediately in the traffic lane due to an error than it is to continue despite the malfunction. In this case, Driving Behaviors can issue a temporary urgency override, disabling certain

error-related software pauses at the health monitor. This urgency override remains engaged until the vehicle is safely within its own travel lane, or a timeout expires.

2.5 Communications

The primary communications backbone on Odin is a switched Gigabit Ethernet network for inter-process communication between software modules, with an additional CAN network for low-level communications confined to the Vehicle Interface. In addition, most sensors were also attached to the Ethernet network, either natively (the IBEO ECUs), or via MOXA Serial-to-Ethernet converters.

2.5.1 JAUS Implementation

For inter-process communication between software modules, Odin employs SAE AS-4 JAUS (joint architecture for unmanned systems). Each of the major software modules (road detection, Motion Planning, health monitor, etc.) is implemented as a stand-alone JAUS component/service that interacts with other software by passing JAUS messages via the local JAUS node manager for that particular computing node (Linux virtual machine, Windows server, CompactRIO, TouchPanel, or developer's laptop), which routes all message traffic within the computing node or to other node managers within the vehicle subsystem.

Components automatically send and receive the JAUS core messages to discover other modules, determine their functionality, and request information from them via periodic or change-based events, enabling dynamic configuration. This allows a software module to be run on a developer's laptop connected to the vehicle network or to be seamlessly shifted to another computing node. In addition, a new module requiring information from an existing module could simply establish a periodic event, without requiring recompilation or reconfiguration of existing software. Figure 12 shows the core message sequence that allows for dynamic configuration to take place. In this example, Driving Behaviors is able to automatically discover localization and create an event to receive local position at a rate of 10 Hz. Although it would be possible to implement the architecture using another messaging structure, the autodiscovery and autoconfiguration features of JAUS greatly eased implementation of the architecture.

Where possible the platform messages in the JAUS standard, such as report velocity state, were used to allow for future interoperability with other JAUS-enabled percepts, planners, platforms, payloads, or control stations. Due to the cutting-edge nature of the urban driving problem JAUS does not yet provide standard messages for everything, and it was necessary to implement many experimental messages. Approximately, 40 experimental messages were developed to pass data such as lane position, static obstacles, dynamic obstacles and predictions, health monitoring errors, turn signal commands, and other data related

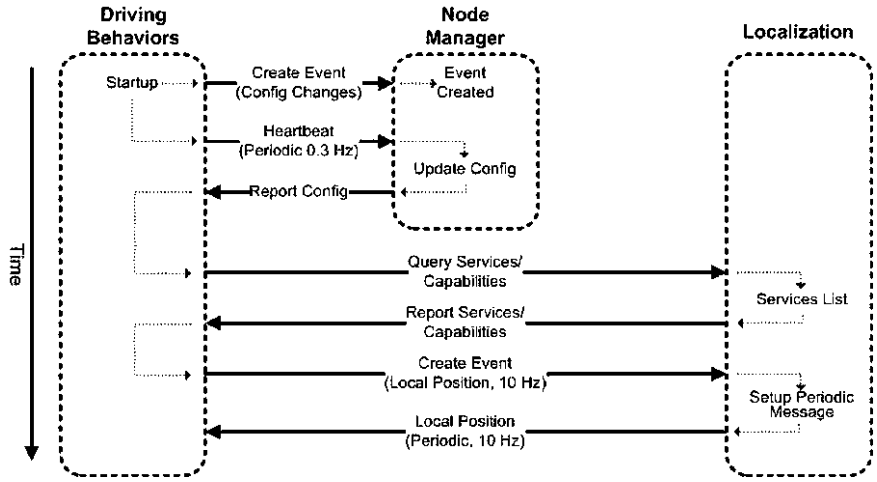


Fig. 12 Detailed message flow during dynamic configuration

to urban driving. With active developers in the SAE AS-4 standards development committee, *Victor*Tango is working to integrate a subset of these messages and protocols into a future version of JAUS.

The JAUS implementation on Odin made use of a well-tested JAUS toolkit previously developed by TORC. In doing so, the communications infrastructure provided key development features such as a generic integrated data-logging and replay system. Additionally, the use of JAUS facilitated scalable simulation tests, from the full software suite to a condensed version running on a single laptop computer. Most importantly, the use of JAUS coupled with careful architecture design allows for the rapid, immediate reuse of Odin's software on other autonomous vehicle projects, a step critical to the acceleration of unmanned systems development.

2.5.2 Integrated Simulation

Simulation played a crucial role in the software development process of Odin, especially in the development and testing of the Planning software. As such, the communications implementation for Odin was designed to facilitate simulation. A custom simulation tool, shown in Fig. 13, was developed using the same messaging architecture as Odin. This tool, combined with the automatic configuration afforded by the JAUS implementation, allowed software components to be transitioned from simulation testing to deployment on Odin without any configuration changes. During full-scale simulation testing, all sensing and perception software messages were generated by the simulator, and Planning software would be run as normal. To isolate planning problems from perception bugs, the simulator produces perfect sensing data using the architecture-defined sensor-independent

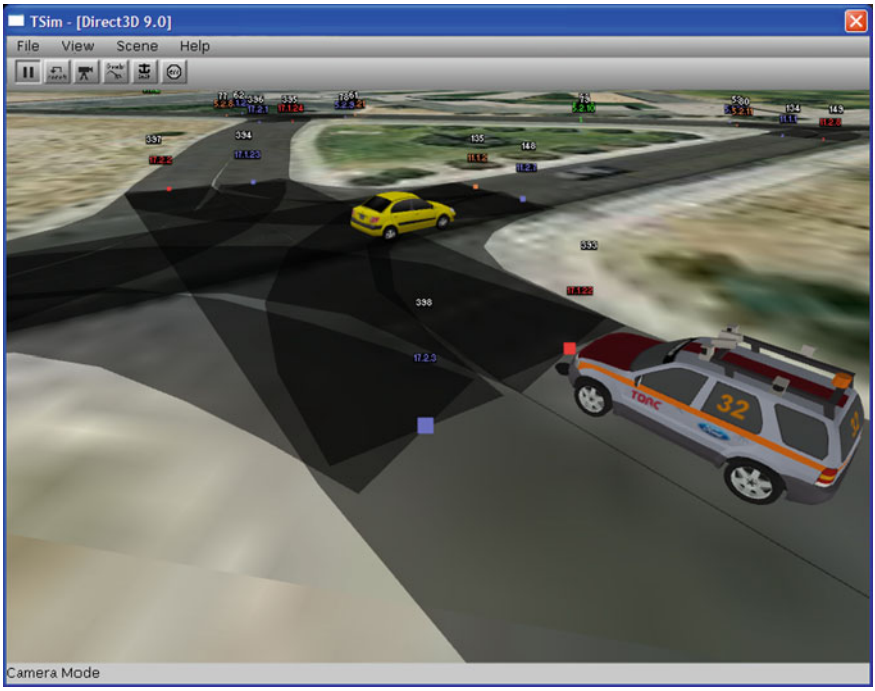


Fig. 13 TSim simulator showing Odin at an intersection

perception messages to report data such as position, static obstacles, and dynamic obstacles. Successful decisions made by the Planning software in the simulation environment would not guarantee success in the real world, however any failures experienced in simulation would most certainly show up during more time-intensive real-world testing.

The simulator was designed to be modular, allowing different vehicle types to be equipped with custom-defined sensors. This modularity allowed for the creation of a vehicle type that directly receives position and orientation updates from the localization software on Odin rather than simulating the motion of Odin. This would allow the use of the simulator as a visualization tool during tests. The same updated vehicle model could also be equipped with virtual sensors just as the fully-simulated vehicle model. This allowed hardware in the loop simulation tests, where Odin would drive in an empty parking lot with no obstacle detection software running. Instead, the simulator would supply static and dynamic obstacles, providing a safe way to test dangerous situations such as traffic not obeying intersection precedence, or minimum stopping distance. Early in software development, any new situation involving traffic would be tested in this manner before attempting with human traffic drivers.

3 Results and Conclusions

The *V*ictor*T*ango software architecture was thoroughly tested and implemented on Odin, one of 35 vehicles invited to participate in the DUC National qualifying event. From these 35, eleven teams were selected to participate in the Urban challenge event. Only six teams managed to complete the course, with the top three places going to Tartan racing, Stanford racing team, and *V*ictor*T*ango, respectively. During the competition, Odin ran for several hours without any human intervention, negotiating stop-sign intersections, merging into and across traffic, parking, and driving down multi-lane roads in the presence of forward and oncoming vehicles. The *V*ictor*T*ango architecture handled all these technical challenges well, proving its overall merit as a field-capable architecture.

Beyond the DUC performance metric, this section will examine more closely the performance of the *V*ictor*T*ango architecture. Several common situations will be examined to demonstrate the flow of information and sequence of decision-making across software modules. An analysis of timing and message latency and their effect on vehicle reaction times will be given, followed by a discussion of future work and areas for improvement.

3.1 Situational Examples

The interactions of the multitude of different software modules on Odin are illustrated here in five simple examples: normal driving on an empty road, handling an intersection, encountering a blocked road, navigating a zone, and undergoing a hardware error. Each example showcases a different type of interaction.

3.1.1 Driving a Road

Before driving, an RNDF and MDF are loaded via a touch-screen user interface located in the vehicle. Upon receipt of the MDF, Driving Behaviors locates itself in the RNDF, chooses a waypoint to begin the route from, and requests a route (in the form of exits and entrances) from the Route Planner. Driving Behaviors then fills in intermediate waypoints between the exits and entrances specified by the Route Planner, pre-assigning a driving situation to each point. Driving Behaviors then spools a small set of waypoints from this pre-determined route to Motion Planning, along with a set of constraints such as desired lane, desired speed, direction, etc. As Motion Planning achieves the commanded points, it reports them achieved and Driving Behaviors shifts the spool forward, activating more specific specialized behaviors depending on the driving situation, to handle traffic or blockages requiring departure from the predetermined lane. These drivers simply override the default behavior by modifying the lane, speed, direction, or target points, and may on occasion request a new route from the Route Planner.

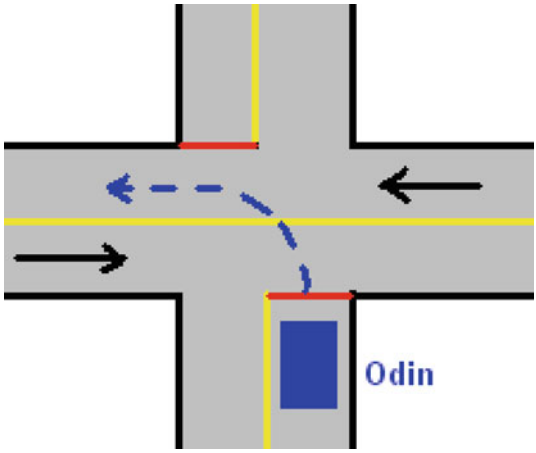


Fig. 14 Intersection with precedence and merging

3.1.2 Intersection

Nearly every software module plays an important role at intersections. On the perception side, road detection describes the form of nearby lanes and their branches through the intersection, object classification identifies and classifies dynamic and static objects, and the dynamic object predictor predicts likely paths for nearby dynamic objects. For planning, Driving Behaviors examines the dynamic objects and predictions to properly observe stop-sign precedence and merge into or cross moving traffic lanes, respecting traffic laws while handling abnormal traffic behavior as necessary. Finally, Motion Planning is responsible for achieving the directives from Driving Behaviors and reporting when its commanded goals are not achievable, while providing basic reflexive avoidance of dynamic and static obstacles.

In the example of approaching a stop sign at an intersection with both through traffic and stop-sign precedence (see Fig. 14), the following sequence of interactions take place. As Odin approaches the intersection, a precedence driver and merge driver are activated within Driving Behaviors; if Odin is making a turn, Driving Behaviors instructs the Vehicle Interface to activate the turn signals. When Motion Planning achieves the stop point and reports it achieved, Driving Behaviors begins waiting for any other vehicles at other stop signs. Once it is Odin's turn to proceed, Driving Behaviors instructs the perception modules (road detection and object classification) to extend their range along the roadway, depending on the direction of approaching traffic. After a safe opening into traffic has been found, Motion Planning is given a non-zero speed, and begins to traverse the intersection. To prevent the health monitor from pausing the vehicle due to an error while crossing a traffic lane, a temporary urgency override is issued to the health monitor once the vehicle is moving fast enough that it cannot safely stop without protruding into the lane. At this point, the vehicle has committed to completing the merge, and the perception modules return to normal sensing range. Finally, once Motion

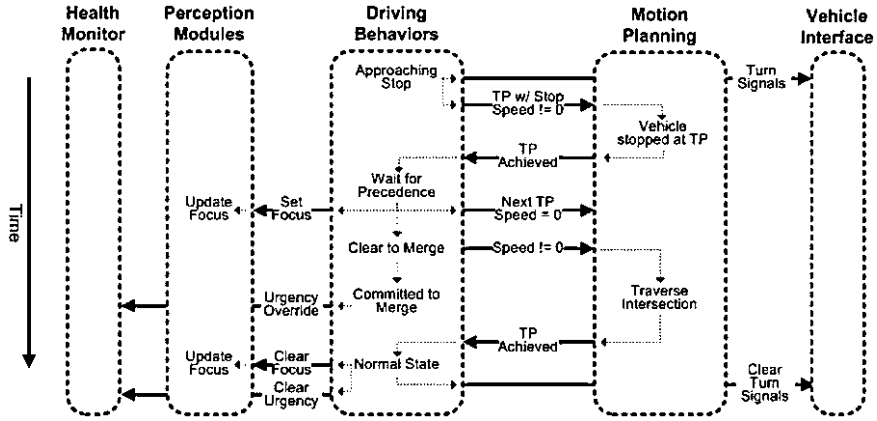


Fig. 15 Messaging sequence when merging at an intersection

Planning reports the next target point achieved, the vehicle returns to its normal road driving situation. This messaging sequence can be seen in Fig. 15.

3.1.3 Blocked Road

In this example, Odin encounters a blockage spanning all available travel lanes of the current RNDF segment. Since only Motion Planning knows the exact kinematic constraints of the vehicle, it is responsible for determining that Odin cannot proceed in the lane being commanded by Driving Behaviors, stopping the vehicle, and reporting the target points as unachievable. A blockage driver within Driving Behaviors monitors the reported achievability from Motion Planning; once the lane is unachievable for sufficient time, Driving Behaviors will try to command a lane change. If all available lanes are reported unachievable and a U-turn is legal, Driving Behaviors alerts the Route Planner of the blockage and requests a new route. After adjusting the graph, the Route Planner determines a new route to achieve the remaining checkpoints in the MDF. Once a new route, complete with a U-turn, has been assembled in Driving Behaviors, the direction is constrained to "don't-care", and Motion Planning is entirely responsible for getting Odin into the new lane, but is free to examine paths in reverse. After Motion Planning completes the U-turn, it reports the goal achieved and Driving Behaviors returns to the default driving situation, with direction "must be forward". This messaging sequence can be seen in Fig. 16.

3.1.4 Parking Lot

The zone navigation problem presents a very different challenge from navigating lanes; however Odin handles it in a similar fashion. One primary difference is that

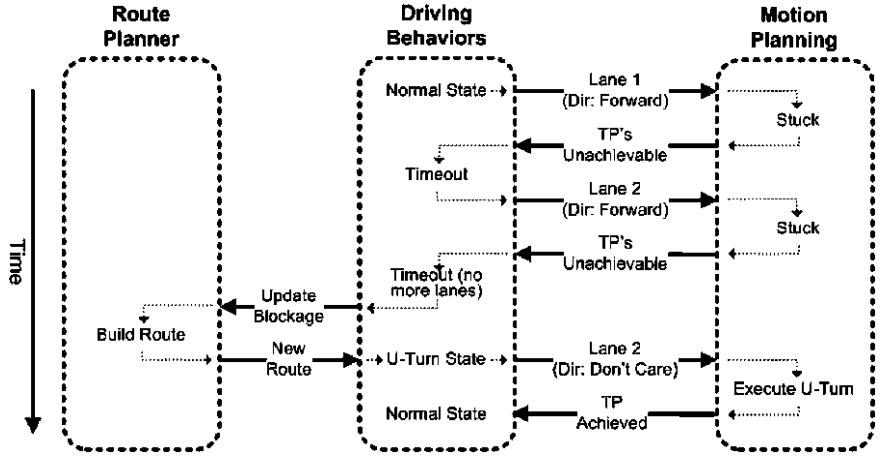


Fig. 16 Messaging sequence when encountering a roadblock

Motion Planning alone is responsible for all dynamic obstacle avoidance in zones, a decision made due to the largely unknown environment in a zone, and the desire to keep Driving Behaviors independent of the exact mobility constraints of the vehicle. Motion Planning also makes more extensive use of the drivable area map from road detection, for avoidance of islands.

To navigate through the zone, to parking spots or the zone exit, target points with constraints are spooled from Driving Behaviors based on a graph of the accepted travel patterns within a parking lot (around parking rows and islands). Motion Planning is then responsible for following these goal points in similar fashion to a roadway lane, with the additional freedom to move in reverse. A robust set of gracefully degrading, increasingly generous recovery modes ensures that Motion Planning will almost always never stop permanently, however without knowledge of the RNDF except as described by road detection, it is unaware of alternate paths to the exit or parking spot. To address this, a progress monitor within Driving Behaviors tracks the vehicle's progress and reported achievability from Motion Planning, and if necessary, can disconnect a segment of the graph and re-plan through the zone.

3.1.5 Hardware Error

One key example of health monitoring manifested itself during the urban challenge. During testing it was observed that the IBEO sensors would occasionally stop sending data and to fix the problem the sensor's ECU had to be restarted. While this 90-s operation is taking place, Odin's ability to perceive objects is severely limited, making it unsafe to continue. This event also occurred during the DUC.

The object classification module detected the error and reported it to the health monitor. At this point, the Driving Behaviors and Motion Planning components had no knowledge of the error and would issue commands as if objects were not present. To prevent this, the health monitor activated a dedicated message which commanded the Vehicle Interface to pause Odin until the sensor had been restarted and the error cleared. Since the urgency override was not activated at the time, the Vehicle Interface brought the vehicle safely to a stop and activated the emergency flashers until the error had cleared. After the sensors reset, Odin was able to proceed with the mission with no human interaction.

3.2 Analysis of Latency and Timings

When designing a software architecture for a task as complex as the urban challenge, care must be taken to ensure that critical information such as the location of obstacles reaches the Planning software before it is too late to react. This is especially important for Odin since all sensor data is processed into sensor-independent messages, adding additional layers of processing and latency. Additional latency is introduced by the DUC software implementation of multiple asynchronous components with fixed processing loop and message rates. The only mechanism in the architecture for synchronization between components are message time stamps inserted into the messages by the sending components. While this feature aids in improving flexibility and implementation speed, additional latency can be introduced when a message is forced to wait for processing at the receiving component until that component's loop recycles.

Messaging in the DUC implementation of the architecture used the JAUS protocol which allows the specification of messaging rates for periodic data. Due to limitations on development time and network bandwidth, messaging rates were determined by setting the rate of transmission for each message was to 10 Hz initially, and increasing the rate as necessary until acceptable performance was achieved. Table 2 presents the processing cycle time of major software components and the messaging rate used per software output. This table is not exhaustive and omits messages such as turn signal control or health monitoring messages. Messages that have a messaging time of 0 ms are ones that are sent immediately, rather than read and forwarded at a periodic rate.

The communications latency has the strongest impact on vehicle's reaction time to static and dynamic obstacles. The reaction time of Odin to a static obstacle, a dynamic obstacle, and intersection behavior is presented in Table 3 with results for best and worst case. The timing variability comes from the system of asynchronous loops. The best case assumes that each module receives the latest inputs just before the processing cycle begins. The worst case assumes that the latest inputs containing a newly sensed obstacle arrive just after the inputs are read, requiring processing to finish and then the loop to run again incorporating the new data. In practice, the expected latency should be in the middle of the best and worst cases.

Table 2 Software latency and timing by message

Software module	Processing time (ms)	Data	Destination	Messaging time (ms)
Vehicle Interface	10	Wheel speed	Localization	10
		Steering angle	Localization	10
			Motion Planning	40
Localization	20	Local position	Vehicle Interface	40
			Report lane position	50
			Motion Planning	33
			Object classification	100
			Driving Behaviors	100
		Velocity state	Vehicle Interface	40
			Motion Planning	33
			Driving Behaviors	100
Road detection, RLP	200	Lane position	Dynamic obstacle predictor	100
			Driving Behaviors	100
			Motion Planning	100
Road detection, DRAC	400	Drivable Area Coverage	Object Classification Motion Planning	100 200
Object classification	100	Static obstacles	Motion Planning	100
		Dynamic obstacles	Dynamic obstacle predictor	100
		Blind Spot status	Driving Behaviors	500
Dynamic obstacle predictor	50	Dynamic obstacle predictor	Motion Planning Driving Behaviors	100 100
Route Planner	Variable	Planned route	Driving Behaviors	0
Driving Behaviors	91	Update RP	Route Planner	0
		Check Point achieved	Route Planner	0
		Behavior profile	Motion Planning	100
MP Speed Limiter	200	Speed Limit	MP Trajectory Search	0
MP Trajectory Search	80 (typ.) 450 (max)	Motion profile	Vehicle Interface	0
		Target Point Achievable	Driving Behaviors	200

The latency from data transfer rates and the non-deterministic nature of Ethernet has been omitted as neither is not a significant factor of latency in the system.

Some effects of latency can be mitigated by the choice of the message data used. For example, static obstacles are transformed from the moving vehicle frame to a stationary locally referenced frame before transmission. Once in the local reference frame, delays in message transmission to and processing by Motion Planning will not affect accuracy. Due to the stationary nature of the local frame, the obstacles will be plotted in the correct positions relative to the latest position of the vehicle established by localization pose messages. The localization data could

Table 3 Total reaction time in common situations

Critical dataflow	Fastest time (ms)	Slowest time (ms)
Response to a static obstacle		
Object Classification	100	298
Motion Planning	80	159
Vehicle Interface	10	19
Total	190	476
Response to a dynamic obstacle		
Object Classification	100	298
Dynamic Obstacle Predictor	50	298
MP Speed Limiter	200	399
MP Trajectory Search	80	159
Vehicle Interface	10	19
Total	440	1073
Decision to proceed at intersection		
Object Classification	100	298
Dynamic Obstacle Predictor	50	198
Driving Behaviors	91	280
MP Speed Limiter	200	399
MP Trajectory Search	80	159
Vehicle Interface	10	19
Total	531	1353

be as much as 99 ms old when it reaches object classification. This could displace sensed obstacles by as much as 0.99 m in the opposite direction of travel, when traveling at 10 m/s. The net result of this transformation is that the geometries are preserved as well as possible but the latency may still affect the maximum distance at which the software can react to obstacles as the vehicle may be closer to the obstacles by time the message is decoded.

The software reaction time and other latency effects could be reduced in the future, but were adequate for the urban challenge competition. Any error in the position of sensed obstacles due to the age of localization data reaching object classification is dependent on Odin's speed, and the error makes objects appear closer to Odin. This will not adversely affect safety since it will cause Odin to begin avoiding an obstacle earlier, and safety buffers in Motion Planning will prevent Odin from clipping the back of an obstacle. The reaction times to static and dynamic obstacles are only significant when new obstacles are sensed, or if data has been updated with a change in position of a static obstacle or a deviation from prediction for a dynamic obstacle. The sensing range and coverage of Odin is sufficient to largely compensate for these negative impacts of latency by ensuring that most obstacles will be detected at a sufficient distance to allow for sufficient reaction time even when the latency effects are included. During development testing and the Urban Challenge, latency effects never caused Odin to collide with a static obstacle.

The reaction time when deciding to proceed at an intersection (stop sign or merging) was also sufficient for the urban challenge, but it causes the vehicle extra delay when traveling and may prevent Odin from merging into a smaller vehicle gap. The urban challenge rules only penalize a vehicle for failing to merge into a vehicle gap of greater than 10 s, allowing Odin to still merge in time. If traffic is detected late and the increased reaction time for merging causes Odin to merge in front of traffic, the software can cancel a merge and hopefully prevent a collision. This type of reaction was demonstrated on Course A in the urban challenge qualifying when Odin successfully canceled a merge when an occluded vehicle was detected.

3.3 Future Work

While the *V*ictor*T*ango software architecture performed extremely well in the DAPRA urban challenge, areas for improvement exist that would enhance the performance as well as the portability of the architecture to new applications and domains.

As shown in the discussion of software latency and message timing, one of the primary weakness to a modular architecture is the relatively high latency that may be associated with actions requiring inputs from multiple components. Such delays were not enough to keep Odin from meeting the requirements set forth in the DARPA urban challenge, however it would be advantageous to reduce these delays where possible. It should also be noted that many of the delays are not necessarily an inherent feature of the architecture. Many of the delays discussed in the results section are due to the processing and transmission rates specific to the implementation of components as used in the DUC.

It would be possible to reduce the latency by implementing each of the components with synchronized messaging and data processing threads to significantly reduce the gap between best-case and worst-case timing scenarios. Additionally, it may be possible to transit some of the components to an event-based messaging and processing model to further reduce intra-component latency. For implementations on systems with dynamic rates similar to those in the DUC, these improvements could eliminate much or all of the latency due to component loop synchronization and arbitrary delays in processing and messaging speed. However, the stacked hierarchical nature of the architecture ensures that the latency can never be reduced below the sum of the minimum processing times for the set of components involved in the action. For the DUC implementation, these improvements could result in latencies at or below the best case scenarios presented above.

A large part of Odin's success can be attributed to the team's rigorous and exhaustive testing in both simulation and in the field. While the architecture supported this rapid design cycle development strategy, it would be desirable to incorporate machine learning into the process. Within Driving Behaviors, for

example, individual behaviors and their position in the HSM must be hand-coded. As a result, determining the control policies and parameters built into each state of each behavior is a time- consuming and error-prone process. It would be beneficial to automatically generate or learn behaviors from experience [1, 2]. Specifically, the use of machine learning and optimization techniques could be used to tweak control policies, state transitions, and command fusion parameters within behaviors. This would reduce the need for supervision and intervention by a human overseer in simulation testing and could reduce development man hours. Additionally, the addition of machine learning has the potential to increase the robustness of the system to unforeseen or changing environments. The modular nature of the *V*ictor*T*ango architecture would easily allow for the reimplementation of any of the components to include machine learning without requiring modification to other components.

Another area for improvement lies in development and incorporation of a more general set of error recovery procedures. Within the *V*ictor*T*ango architecture, error recovery of hardware and software happens on many different levels from component-level errors at the health monitor to more specific solutions within individual modules. These error recovery procedures were developed through the testing process as problems arose. This sometimes led to an awkward sharing of error recovery responsibilities between modules. Furthermore, there is no process laid out in the architecture that guarantees that new errors or unforeseen situations are handled in a specific manner. To truly expand the usability of the system to more varied and unstructured environments requires a more cohesive error recovery system that remains distributed, but is more robust and adaptable to unforeseen events.

While the decision to employ sensor-independent perception messages as virtual sensors enhances the portability of the software, the set of information they contained was minimal. It would be beneficial to add a small set of messages to communicate more metadata to the Planning software. For example, Driving Behaviors operates using only the dynamic object list, and with no notion of the sensor modalities or sensor locations used in perception, it cannot accurately reason about occluded areas. While Driving Behaviors incorporated mechanisms to deal with short periods of occlusion, a specialized Occlusion Hypothesizer with knowledge of sensor location and sensor state would make Odin more robust to longer periods of occlusion, such as the one in Fig. 17. Alternatively, the Occlusion Hypothesizer could be incorporated into object detection and a new virtual sensor could be created to report likely occluded areas to Driving Behaviors. The architecture offers the flexibility to implement this feature in either form.

Another sensor-related modification of the *V*ictor*T*ango architecture would be the incorporation of simultaneous localization and mapping (SLAM) algorithms. The modular and separated nature of localization and object detection in the current implementation is antithetical to the inherently related nature of these two functions in a SLAM algorithm. If high enough messaging rates between the components can be achieved, which may be possible due to the asynchronous aspects of the architecture, it may be possible to implement a SLAM algorithm using the existing component structure. Alternatively, the two separate

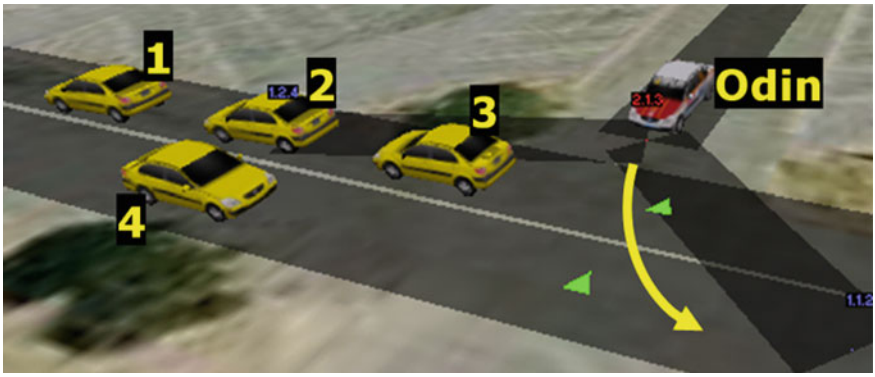


Fig. 17 Odin's view of vehicle 4 is temporarily occluded for a medium period of time

components could be combined into a single component responsible for both tasks. As long as the new component can output the same virtual sensors, this implementation could be transparent to the remainder of the components.

Finally, it may be desirable in the future to add or replace certain components to accommodate the needs of new missions. For example, to enable off-road navigation it may be possible to replace the road detection component with a component responsible for terrain mapping and path detection. If the new component were implemented in such a way as to create messages similar to those produced by road detection (i.e. virtual roads), a new capability could be added with minimal modifications to the remainder of the components or the architectural messaging structure.

3.4 Conclusions

The DARPA urban challenge posed many important technical challenges to the mobile robotics community. The problems of perception, artificial intelligence, navigation, and path planning all had to be addressed and implemented reliably. Beyond these individual solutions, however, a fully-autonomous system must bring these pieces together as parts of a much larger machine. Their interactions must be carefully designed and tested such that the entire system is capable of completing complex, multi-objective tasks in dynamic, unpredictable environments. Furthermore, to complete the urban challenge the system must perform correctly 100% of the time, handling errors or malfunctions internally.

To address this difficult problem, the *VictorT*ango architecture employs a tri-level hybrid deliberative/reactive approach that mimics biological intelligence in some aspects and takes advantage of brute force computing in others. This architecture exhibits a deliberative-reactive-deliberative progression, utilizing optimal planning techniques for high-level mission planning as well as low-level

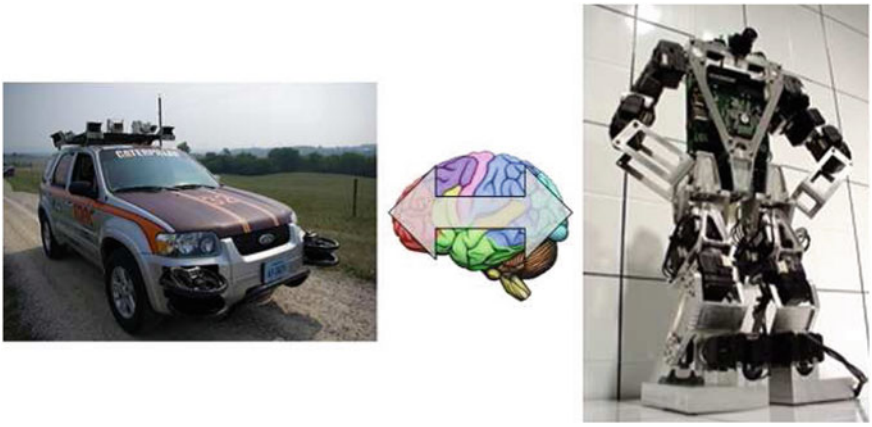


Fig. 18 Application of a portable software architecture across autonomous system domains

Motion Planning. Integration of a deliberative layer at the Motion Planning level is possible due to the development of an efficient, model-predictive search algorithm that takes advantage of pre-computed values and powerful processing capabilities. This addition subsequently shifts the typical scope and application of the behavior-based layer to higher-level decision-making. A novel ASM is used within this reactive layer that takes advantage of both hierarchy and parallelism by placing a behavioral HSM for arbitration in sequence with a command fusion mechanism. As a result, contextual intelligence and emergent behavior are both preserved. Asynchronous perception processing modules continually fuse sensor data and produce a set of virtual sensors describing the environment. Virtual actuators are used to maintain hardware independence and to promote future reusability of software modules in new domains. Finally, the standardized JAUS communications protocol for inter-process messaging is used, allowing for dynamic reconfiguration of computing nodes, portability to new platforms, and the simple incorporation of simulation and HIL testing.

Odin's performance in the DARPA urban challenge proves that the *Victor*Tango architecture is capable of enabling autonomous vehicles to perform complex tasks. The flexibility of the architecture allows room for future improvements and it has also been ported to new vehicles and autonomous systems as illustrated in Fig. 18. By leaving room for modifications and improvements within software modules, the *Victor*Tango architecture has proven adaptable to several different applications. One example of porting this approach to a drastically different domain has been on DARwIn [11, 16], a bi-pedal, 2-foot tall humanoid robot capable of playing soccer. While a new set of experimental messages, behaviors, Motion Planning, and perception algorithms were developed, the fundamental problems of autonomy remain the same and the benefits of the *Victor*Tango architecture still apply. Additionally, the architecture has been implemented successfully on a range of

other autonomous and semi-autonomous vehicles to solve real-world problems significantly less constrained than the urban challenge. The flexibility of the *V*ictor*T*ango architecture and the continued and rapid development of autonomous technology indicate that the architecture is likely to be ported to many other platforms and domains in the future.

Acknowledgments Many people and organizations contributed greatly to the success of team *V*ictor*T*ango. The authors would first like to thank DARPA for creating and sponsoring the grand challenge series that motivated this work. Next, we would like to thank the other members of team *V*ictor*T*ango, especially the remaining graduate students: Tom Alberi, Dave Anderson, Aaron Dalton, Mike Webster, and David VanCovern as well as the long list of undergraduate students who contributed to this project. Finally, we would like to thank our sponsors, especially our premier sponsors: Caterpillar, Inc. and Ford Motor Co. This project would not have been possible without their support or the hard work of the many members of the *V*ictor*T*ango team.

References

1. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Progr. Lang Syst. **23**(3), 273–303 (2001)
2. Argall, B., Browning, B., Veloso M.: Learning to select state machines using expert advice on an autonomous robot. In: EEE International Conference on Robotics and Automation, pp. 2124–2129 (2007)
3. Arkin, R.C.: Motor schema based navigation for a mobile robot: an approach to programming by behavior. In: IEEE International Conference on Robotics and Automation, pp. 264–271 (1987)
4. Arkin, R.C., Riseman E.M., Hansen, A.: Aura: an architecture for vision-based robot navigation. In: DARPA image understanding workshop, pp. 413–417. Los Angeles, CA (1987)
5. Bacha, A., Bauman, C., Faruque, R., Fleming, M., Terwelp, C., Hong, D., Wicks, A., Reinholtz, C., Alberi, T., Anderson, D., Cacciola, S., Currier, P., Dalton, A., Farmer, J., Hurdus, J., Kimmel, S., King, P., Taylor, A., Covern, D.V., Webster, M.: Odin: Team victortango's entry in the darpa urban challenge. J. Field Robot. **25**(8), 467–492 (2008)
6. Brooks, R.A.: A robust layered control system for a mobile robot. IEEE J. Robot. Autom. **2**(1), 4–23 (1986)
7. Bryson, J.: From animals to animats 6(SAB00), chap Hierarchy and Sequence vs. Full Parallelism in Reactive Action Selection Architectures, pp. 147–156. MIT Press, Cambridge (2000)
8. Cacciola, S.: Fusion of laser range-finding and computer vision data for traffic detection by autonomous vehicles. Master's Thesis, Virginia Tech, Blacksburg, VA (2007)
9. Currier, P.: Development of an automotive ground vehicle platform for autonomous urban operations. Master's Thesis, Virginia Polytechnic Institute and State University (2008)
10. Gat, E.: Artificial Intelligence and Mobile Robots, chap Three-layer Architectures. MIT Press, Cambridge (1998)
11. Hurdus, J.G.: A portable approach to high-level behavioral programming for complex autonomous robot applications. Master's Thesis, Virginia Tech, Blacksburg, VA (2008)
12. Kelly, A.: A 3d state space formulation of a navigation kalman filter for autonomous vehicles. Technical Report CMU-RI-TR-94-19, CMU Robotics Institute (1994)
13. Khatib, O.: Real-time obstacle avoidance for manipulators and mobile robots. Int. J. Robot. Res. **5**(1), 90–98 (1986)

14. Konolige, K., Myers, K.: Artificial Intelligence and Mobile Robots, chap The Saphira Architecture for Autonomous Mobile Robots. MIT Press, Cambridge (1998)
15. Minsky, M.: The Society of Mind. Simon and Schuster, New York (1985)
16. Muecke, K., Hong, D.: Darwins evolution: development of a humanoid robot. In: IEEE International Conference on Intelligent Robotics and Systems (2007)
17. Murphy, R., Mali, A.: Lessons learned in integrating sensing into autonomous mobile robot architectures. J. Exp. Theor. Artif. Intell. **9**(2), 191–209 (1997)
18. Murphy, R.R: Introduction to AI Robotics. MIT Press, Cambridge (2000)
19. Pirjanian, P.: Behavior coordination mechanisms state-of-the-art Tech Report IRIS-99-375 Institute for Robotics and Intelligent Systems. University of Southern California, Los Angeles, California (1999)
20. Russel, S., Norvig, P.: Artificial Intelligence A Modern Approach Pearson Education Inc., Upper Saddle River (2003)
21. Simmons, R., Goodwin, R., Haigh, K., Koenig, S., O'Sullivan, J.: A layered architecture for office delivery robots. Autonomous Agents. **97**, 245–252 (1997)
22. Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A., Mahoney, P.: Stanley: The robot that won the darpa grand challenge: Research articles. J. Fld. Robot. **23**(9), 661–692 (2006)
23. Urmson, C., Anhalt, J., Bartz, D., Clark, M., Galatali, T., Gutierrez, A., Harbaugh, S., Johnston, J., Kato, H., Koon, P.L., Messner, W., Miller, N., Mosher, A., Peterson, K., Ragusa, C., Ray, D., Smith, B.K., Snider, J.M., Spiker, S., Struble, J.C., Ziglar, J., Whittaker, W.R.L: A robust approach to high-speed navigation for unrehearsed desert terrain. J. Fld. Robot. **23**(8), 467 (2006)
24. Webster, J.: A localization solution for an autonomous vehicle in an urban environment. Master's thesis, Virginia Tech, Blacksburg, VA (2007)

Real-time Obstacles Avoidance for Vehicles in the Urban Grand Challenge

Jian Yang, Jing Wang, Mi Dong and Zhihua Qu

Abstract The aim of this chapter is to present a real-time trajectory planning solution for vehicles in the urban grand challenge. Typically, reference paths generated by a high-level path planner are not feasible for vehicles and need improvement. In addition, moving obstacles in the environment are generally not known a priori, which also requires the paths to be able to be replanned in real-time. The proposed method presents three novel features to satisfy those requirements. First, all the paths satisfying boundary conditions and the vehicle's kinematic constraints are parameterized in terms of polynomials of sufficient order. Then, obstacles are classified to two types: "hard" obstacles that must be avoided, and "soft" obstacles that can be run over/through. And a collision-free criterion is developed for avoiding these obstacles detected. In the third step, a L_2-norm performance index is introduced to find the best path among the class of collision-free paths. The performance index is chosen such that a path analogous to the shortest path can be analytically solved. Optimal feasible trajectories are analytically solved for "hard" obstacles. By relaxing the optimal solution, "soft" obstacles are prioritized to be bypassed or overcome. Moreover, techniques to further improve the performance are discussed. The proposed method provides a systematic and analytical solution to find the feasible path while addressing

J. Yang (✉) · Z. Qu
University of Central Florida, Orlando, FL 32816, USA
e-mail: fish2bear@hotmail.com

Z. Qu
e-mail: qu@mail.ucf.edu

J. Wang
Bethune-Cookman University, Daytona Beach, FL 32114, USA
e-mail: wangj@cookman.edu

M. Dong
Central South University, Changsha, Hunan 410083, China
e-mail: dongmi@gmail.com

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_6, © Springer-Verlag London Limited 2012

obstacle avoidance and guaranteeing performance. Simulation results verify the proposed algorithm.

1 Introduction

TeamUCF and the Knight Rider vehicle were conceived over three years ago at the beginning of the 2005 DARPA Grand Challenge event. With the inception of the DARPA Urban Challenge, TeamUCF has built on the existing capabilities of the Knight Rider vehicle, creating a competition vehicle with the goal of meeting all the mission objectives of DARPA and performing at a level that challenges all other entries. Our Urban Challenge vehicle is known as Knight Rider VIP and focuses on the implementation of higher-level capabilities through the use of Vision, Intelligence, and Planning in a technical and motivational sense. Computer vision systems utilize a combination of sensors including active laser scanners, visible spectrum passive cameras, and near field acoustic sensors. Intelligence systems include a context-based reasoning subsystem which build a real-time environmental model augmenting the supplied route network definition file (RNDF) and develops mission plans to achieve objectives identified in the mission data file (MDF). Planning provides tactical guidance information and vehicle control to achieve the defined missions in a safe and effective manner.

The most interesting element of planning is perhaps the generation of dense path information from the relatively sparse information provided by Intelligence. The fundamental objective of this element is to develop a real-time path planning and control algorithm by which the vehicle can autonomously navigate through a dynamically changing and congested environment. In the urban scenarios the vehicle must operate within very tight space, must take motion constraints into consideration (including boundary condition, kinematic constraints, and velocity bounds), must avoid static as well as moving obstacles, and must perform complicated maneuvers whenever necessary.

A lot of research efforts have been directed toward motion planning in the presence of static obstacles, and many techniques have been proposed. Among them, the method of potential field pioneered in [1] is widely used, and its basic idea is that, in planning a trajectory, potential fields are built around obstacles and pathways to expel the trajectory from obstacles and to bring the trajectory close to the final destination. Follow-up work can be found in some papers [2–5]. However, this methodology does not explicitly consider the system's kinematic constraints, and the generated trajectories may not be feasible. In order to solve this problem, some remedies have been proposed in the literature by first generating the paths without considering kinematic constraints and then accommodating the feasibility segmentally. For instance, Sundar [6] gave the online suboptimal obstacle avoidance algorithm and Laumond [7, 8] presented the nonholonomic path planner using a sequence of optimal path segments. Also popular are the spline methods

[9–11], in which a sequence of splines are used to generate a path through a given set of waypoints in the environment. However, the obtained trajectory is fixed and may not be collision-free for a dynamically changing environment. This makes it less interesting for real-time trajectory planning. Nonetheless, its underlining idea of parameterization and optimization is quite general and useful.

Exhaustive search and numerical iteration methods have also been used to deal with kinematic constraints and collision avoidance. By dividing the space into a series of regions, Barraquand [12] found a safe path for the vehicle by starting at the initial condition and successively searching adjacent regions to the goal. The path segments are constructed by discretizing the controls and integrating the equations of motion. In Wen's paper [13], the obstacle avoidance criterion and kinematic model are typically converted into a set of inequality and equality constrains, and numerical iterations are used to approximate or determine the path that satisfies all the constraints.

Few results are available to deal with the moving obstacles. Kant decomposed the dynamic motion planning to a static trajectory planning problem and a velocity planning problem [14]. Erdmann treated the time as a state variable and recasted the dynamic motion planning problem into a static one [15], and so on. However, these approaches suffer from incomplete information and their solutions are not guaranteed.

To solve the real-time trajectory planning problem in a dynamic environment, new methodologies are presented in the recent work [16], an analytic solution was presented to the problem of determining a collision-free trajectory for a car-like vehicle moving in a dynamically-changing environment. A class of collision-free trajectories in a closed form was determined by one adjustable parameter. Unfortunately, the paper [16] did not talk about the optimal solution of the trajectory. Thus the obtained trajectory sometimes is not reasonable. The paper [17] addressed an optimal solution, however it did not consider different types of obstacles, this stiff framework sometimes cannot guarantee a good result. Moreover, these two papers treated the obstacles as circles with different sizes, thus, these approaches will lose some generalities.

Recently, to solve the real-time trajectory planning problem for a car-like vehicle moving in a dynamic environment, a new method is presented by Qu and Wang [16]. A class of collision-free trajectories in a closed form is analytically determined by one adjustable parameter. However, no optimal performance is considered in the solution. Thus the obtained trajectory may not be efficient in certain contexts. More recently, as a continuation of the work [16], an optimal solution is obtained in Yang's paper [17]. However it does not explicitly address the more realistic situation with different types of obstacles such as "hard" and "soft" obstacles, which may call for different criterion for more efficiently generating optimal trajectories.

This chapter originated from the work in [18] and gives more general results. It is organized as follows. In Sect. 2, the problem is rigorously formulated. Particularly, we first give the mathematical models for vehicle and dynamical environments. Then, the family of trajectories in terms of parameterized polynomials is

presented. Third, an optimal performance index in the form of L_2 norm is defined for trajectory planning. Finally, we introduce the collision-avoidance criterion. In Sect. 3, an analytical and optimal collision-free solution for both "hard" and "soft" obstacles are obtained, and the design process is illustrated step by step. Also discussed are the approach to improve the performance for trajectories obtained. Extensive simulations are performed to validate the effectiveness of the proposed approach in Sect. 4. The conclusion is drawn in Sect. 5.

2 Problem Statement and Mathematical Formulation

In this chapter, an optimal and real-time trajectory planning problem is formulated and solved. At the abstract level, the general problem of optimal trajectory planning is to determine $\mathbf{q}(t) : [t_0, t_f] \mapsto \mathbb{R}^{N_q}$ which meets the following sets of conditions:

C.1: Boundary conditions at the initial and terminal time instants, that is,

$$\mathbf{q}(t_0) = \mathbf{q}_0, \quad \mathbf{q}(t_f) = \mathbf{q}_f, \quad t_f > t_0; \tag{1}$$

C.2: Motion constraints in the form of

$$H(\dot{\mathbf{q}}(t), \mathbf{q}(t), \mathbf{u}(t)) = 0, \quad \forall t \in (t_0, t_f) \tag{2}$$

for some control inputs $\mathbf{u} \in \mathbb{R}^{N_u}$;

C.3: Minimization of performance index

$$J[\mathbf{q}(\cdot), \mathbf{u}(\cdot), t_0, t_f] = \int_{t_0}^{t_0+T} W(\mathbf{q}(t), \mathbf{u}(t))dt; \tag{3}$$

C.4: Collision avoidance, i.e., for any point $\mathbf{q}_{obs}(t)$ related to any of the obstacles,

$$\|\mathbf{q}(t) - \mathbf{q}_{obs}(t)\| > \rho_o, \tag{4}$$

where N_q is the dimension of the configuration space, $T \triangleq t_f - t_0$, $\rho_0 > 0$ is the minimum measure on physical envelope of the vehicle, and $\|\cdot\|$ denotes an appropriate distance measure. ***Feasible*** trajectories are defined to be ones satisfying conditions **C.1**, **C.2**, and **C.4**.

In real-world applications, the solution to a trajectory planning problem depends on specifics of the above conditions. Several of the conditions, such as boundary condition Eq. 1 and motion constraints Eq. 2, are usually given, nonetheless it is important that a good algorithm is developed to accommodate their variations in different applications. Optimal performance index Eq. 3 and collision-avoidance criteria Eq. 4 are chosen by the designer to obtain the best algorithm possible.

In the rest of the section, specific choices of the conditions are made to illustrate the features and generality of the proposed trajectory planning algorithm.

Fig. 1 The Knight Rider Vehicle

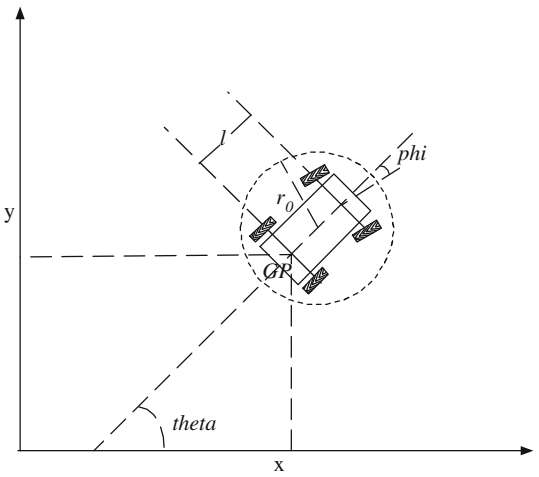
In particular, a kinematic model of a differentially driven vehicle is stated in Sect. 2.1 and is used to illustrate how condition **C.2** is handled in the proposed algorithm. To describe most changing environments in the real world, a piecewise-constant environment model is introduced in Sect. 2.2, and obstacle envelopes are specified. To handle a dynamically changing environment, the class of piecewise-constant and polynomial-parameterized trajectories are defined in Sect. 2.3. Our approach is to determine a real-time, optimal, and feasible trajectory in the class. To this end, condition **C.3** is chosen in Sect. 2.4 to be a L_2-norm-based performance index in order to obtain an analytical solution; and in Sect. 2.5, collision-avoidance criteria are presented to handle obstacles.

2.1 Kinematic Model of the Knight Rider Vehicle

The Knight Rider Vehicle is shown in Fig. 1, it is a front-steering and rear wheel drive vehicle. Let (x,y) be the Cartesian coordinates of the guidepoint (i.e., the middle point of the rear axle), θ the orientation of the car body with respect to the x axis, ϕ the steering angle, ρ the driving wheel radius, l the distance between the two wheel axles, u_1 the angular velocity of the driving wheel, and u_2 the steering rate. By limiting $\phi \in (-\pi/2, \pi/2)$ (or else by applying by a state transformation), we know that its kinematics model is given by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \rho\cos(\theta) & 0 \\ \rho\sin(\theta) & 0 \\ \frac{\rho}{l}\tan(\phi) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}. \tag{5}$$

Fig. 2 A car-like vehicle



It is worth noting that the above model is to illustrate the proposed trajectory planning framework. Should the vehicle's kinematic model be different, the proposed idea would still be applicable. It can be shown that the class of polynomial-parameterized trajectories in Sect. 2.3 are applicable to all vehicles whose kinematic constraints can be transferred into the chain form [19].

As to interact with its environment, the vehicle has a physical envelope which is assumed to be a circle centered at $O(t)$=(x,y) and of radius r_0, and it is equipped with sensors to detect all the obstacles in its sensing range which is assumed to be a circle of radius R_s with $R_s > r_0$ (Fig. 2).

2.2 Dynamically-Changing Environment

Mathematically, a dynamically changing environment can be described by a sequence of piecewise-constant environments, that is, there exists a sampling period $T_s > 0$ such that, within each time interval $t \in [t_0 + kT_s,\ t_0 + (k+1)T_s)$ where $k = 0, 1, 2, \ldots, \frac{T}{T_s}$ and T/T_s is an integer, velocities of obstacles detected are constants. In this chapter, superscript ^{k} and subscript $_k$ both are the index for the kth sampling period, for instance, $t_k = t_0 + kT_s$. The proposed mathematical abstraction of a changing environment consists of the following:

- Vehicle: In addition to its physical model (to be stated shortly), it has a limited sensing range (denoted by a circle of radius R_s) and two-dimensional operational velocity $v(t) \triangleq [\dot{x}, \dot{y}]^T$. Physical envelope of the vehicle is assumed to be a circle centered at $O(t) = (x, y)$ and of radius r_0.
- Obstacles: For some nonnegative integer N_o (which may be unknown), the ith obstacle (for any $i \in \{1, \ldots, N_o\}$) has a circular envelope centered at

$O_i(t) = (x_i, y_i)$ and of radius r_i, and it moves according to its instantaneous velocity $v_i(t) \triangleq [v_{i,x}, v_{i,y}]$.

- Detection: Motion of obstacles are not modeled, known a priori or predictable. Nonetheless, as soon as an obstacle emerges within the sensing range of the vehicle, the corresponding center location $O_i(t) = (x_i, y_i)$, and motion velocity $v_i(t)$ are all detected. At time t, the number of obstacles detected is denoted by $n_o(t)$ (with $n_o(t) \leq N_o$).
- Piecewise-constant evolution: there exists a sampling period T_s such that, within the time interval $t \in [t_0 + kT_s,\ t_0 + (k+1)T_s)$, $v_i(t) = v_i^k$ are constant (for those $n_o(t)$ obstacles seen by the vehicle). Accordingly, trajectory replanning needs to be done continually to take into account new sensing information, for which an analytical solution is preferred for real-time implementation. In the latter parts, the subscript or superscript k of all variables mean they are in the kth sampling period.

Graphically, the operation of the vehicle can be represented by Fig. 3, and the above abstraction accounts for most of the practical issues. In addition, sampling period T_s could be allowed to vary over time (except that notations would become more involved).

2.3 Piecewise Polynomial-Parameterization of Trajectories

As described in Sect. 2.2, the environment is modeled as piecewise-constant evolution. Therefore, the trajectory needs to be evolved piecewise. To meet the requirement of piecewise evolution, in the kth sampling period, the corresponding boundary conditions of trajectories must be satisfied, which are given as

$$\mathbf{q}(t_k) = \mathbf{q}_k = [x_k, y_k, \theta_k, \phi_k], \quad \text{and} \quad \mathbf{q}(t_f) = \mathbf{q}_f = [x_f, y_f, \theta_f, \phi_f], \tag{6}$$

where $\mathbf{q}_k$ is the configuration variable at the beginning of the kth ($k > 0$) sampling period, determined from the trajectory planned in the previous period, or is the initial condition $\mathbf{q}_0 = (x_0, y_0, \theta_0, \phi_0)$.

According to the approximation theorem, trajectories satisfying boundary conditions Eq. 6 can be described by the polynomial shown as follows:

$$y = [a_0^k \ \ a_1^k \ \ \ldots \ \ a_p^k][1 \ \ x \ \ \ldots \ \ x^p]^T, \tag{7}$$

where p (≥ 5) is the order of the polynomial. It is clear that the case of $p = 5$ generates a unique trajectory, which is fixed and gives no freedom to avoid obstacles. $p > 5$ yields a family of trajectories, thus collision-free paths can be chosen from them. Based on kinematic Eq. 5, assuming the total steering time T is given, the general class of trajectories under boundary conditions Eq. 6 and the corresponding control inputs are obtained according to the following lemma.

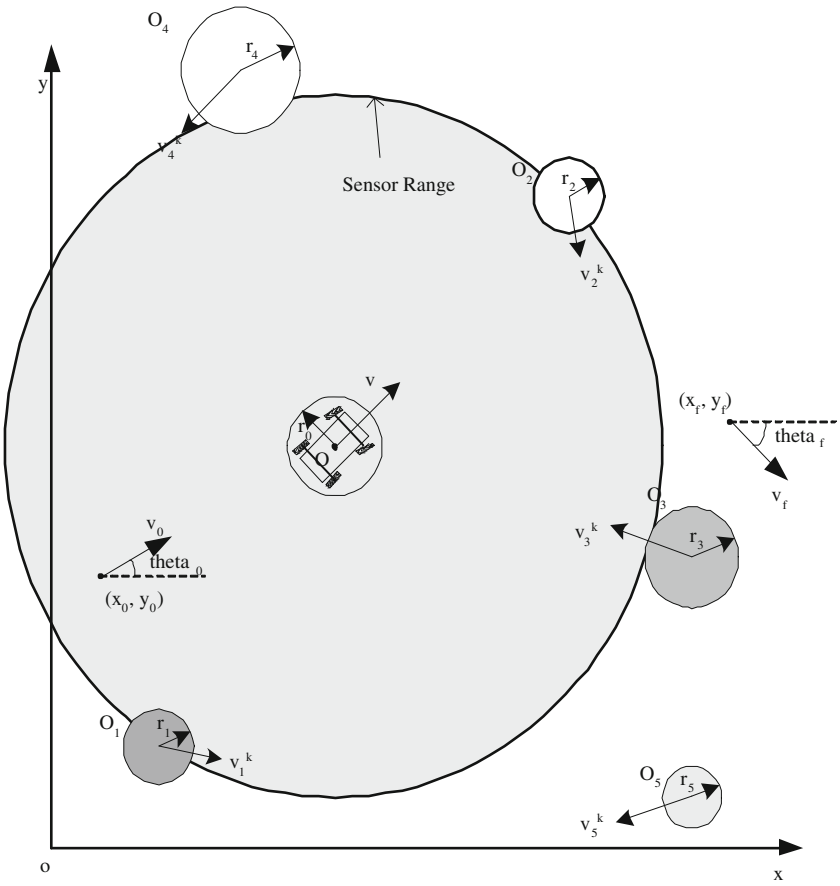


Fig. 3 Operation of a vehicle with limited sensing range

Lemma 0.1 *Consider a mobile vehicle given by Eq. 5. Under the boundary conditions Eq. 6, for* $t \in (t_0 + kT_s, t_0 + (k+1)T_s]$, *the family of the mobile vehicle's trajectories can be parameterized as follows* [16]:

$$y = X[{a_0}^k, {a_1}^k, {a_2}^k, {a_3}^k, {a_4}^k, {a_5}^k]^T + {a_6}^k x^6, \tag{8}$$

and is achievable under the following steering controls:

$$u_1^k = \frac{w_1^k}{\rho \cos(\theta_k)}, \tag{9}$$

and

$$u_2^k = -\frac{3\sin(\theta_k)}{l\cos^2(\theta_k)} \sin^2(\phi_k) w_1^k + l\cos^3(\theta_k)\cos^2(\phi_k) w_2^k, \tag{10}$$

where a_6^k is a free design parameter and

$$
\begin{aligned}
&X = [1\ x\ x^2\ x^3\ x^4\ x^5],\\
&[a_0^k, a_1^k, a_2^k, a_3^k, a_4^k, a_5^k]^T = (B^k)^{-1}(Y^k - A^k a_6^k)\\
&A^k = [(x^k)^6\ \ 6(x^k)^5\ \ 30(x^k)^4\ \ (x_f)^6\ \ 6(x_f)^5\ \ 30(x_f)^4]^T,
\end{aligned}
$$

$$
Y^k = \begin{bmatrix} y^k \\ \tan(\theta^k) \\ \dfrac{\tan(\phi^k)}{l\cos^3(\theta^k)} \\ y_f \\ \tan(\theta_f) \\ \dfrac{\tan(\phi_f)}{l\cos^3(\theta_f)} \end{bmatrix}, \quad
B^k = \begin{bmatrix} X|_{x=x^k} \\ \dfrac{\partial X}{\partial x}|_{x=x^k} \\ \dfrac{\partial^2 X}{\partial x^2}|_{x=x^k} \\ X|_{x=x_f} \\ \dfrac{\partial X}{\partial x}|_{x=x_f} \\ \dfrac{\partial^2 X}{\partial x^2}|_{x=x_f} \end{bmatrix},
$$

$$
\begin{aligned}
w_1^k &= \frac{x_f - x_k}{t_f - t_k},\\
w_2^k &= 6\left[a_3^k + 4a_4^k x_1^k + 10a_5^k (x_1^k)^2 + 20a_6^k (x_1^k)^3\right] w_1\\
&\quad + 24\left[a_4^k + 5a_5^k x_1^k + 15a_6^k (x_1^k)^2\right](t - t_0 - kT_s){w_1}^2\\
&\quad + 60\left(a_5^k + 6a_6^k x_1^k\right)(t - t_0 - kT_s)^2 {w_1}^3\\
&\quad + 120a_6^k (t - t_0 - kT_s)^3 {w_1}^4.
\end{aligned}
$$

Proof Recalling Eq. 5, it is straightforward to obtain that

$$
\frac{dy}{dx} = \tan\theta, \quad \frac{d^2y}{dx^2} = \frac{\tan(\phi)}{l\cos^3(\theta)}, \tag{11}
$$

Taking advantage of this characteristic, we can satisfy both vehicle's kinematic model Eq. 5 and boundary condition Eq. 6 by parameterized polynomials of sufficient order. Actually, to ensure there exist a solution, at least a fifth-order polynomial should be applied. Choosing a six order polynomial of y in terms of x, shown as Eq. 8, it is easy to obtain all the results according to Eq. 11. □

Since $p = 5$ is the simplest case to achieve a family of trajectories, we discuss it to present the framework of avoiding obstacles. Thus, only one adjustable parameter a_6^k is kept and used to generate trajectories. Any collision avoidance criterion for obstacles can be mapped into the domain of a_6^k, therefore feasible trajectories are obtained.

In each sampling (evolution) period, the piecewise sixth-order polynomial obtained from Lemma 0.1 may not represent arbitrary trajectories well. Also, since $w_1 = (x_f - x_k)/(t_f - t_k)$, the obtained trajectory can only move in one direction

along the x axis, which means the obtained trajectory is a single-valued function. However, given waypoints, each pair of neighbor points (including original boundary points and waypoints) and their associated steering angles are treated as new boundary conditions, and Lemma 0.1 is reapplied to generate sub-trajectories. The obtained series of sixth-order polynomials (like interpolation) are still applicable to describe any trajectories. These waypoints are given by the higher-level planner according to requirements. For instance, based on static maps, typical searching approaches such as A^* could be used to find these waypoints by using some prior knowledge.

2.4 Optimal Performance Index for Trajectory Planning

To obtain an optimal solution from the family of trajectories, an applicable performance index Eq. 3, such as the length of path, should be assigned. The method to obtain the shortest trajectory is to compute the minimum value of the line integral with respect to arc length. The solution can only be obtained numerically. However, for applications in dynamical environments, the optimal trajectory should be obtained in real-time. Thus, an analytical solution is preferred. In what follows, we present a performance index in the form of L_2 norm such that an optimal trajectory, analogous to the shortest path, will be achieved. Moreover, the optimal solution is analytical. Intuitively, without taking into account kinematic constraints, it is well known that the shortest trajectory from the starting point to the end point is the beeline between them, called "initial straight line". Hence, if the feasible trajectory for the mobile vehicle, where kinematic constraints are considered, can be made to stay as close as possible to this "initial straight line" in real-time, then it is reasonable to say that the trajectory is shortened effectively. To this end, let the cost function be formulated according to the integral of the square of the difference between the vertical coordinate of the parameterized trajectories and that of the "initial straight line". It is graphically illustrated in the Fig. 4. To match the piecewise-constant evolution in the kth sampling period, the cost function is rewritten as

$$J_k = \int_{x_k}^{x_f} \left[y - \frac{y_f - y_0}{x_f - x_0}(x - x_0) - y_0 \right]^2 \mathrm{d}x. \tag{12}$$

Thus, for this chapter, the optimal problem is to choose feasible trajectories Eq. 8 which minimizes Eq. 12.

2.5 Collision-Avoidance Criteria for Different Types of Obstacles

Generally, two kinds of obstacles exist in the environment, that is, "hard" obstacles and "soft" obstacles. Their definitions are shown as follows:

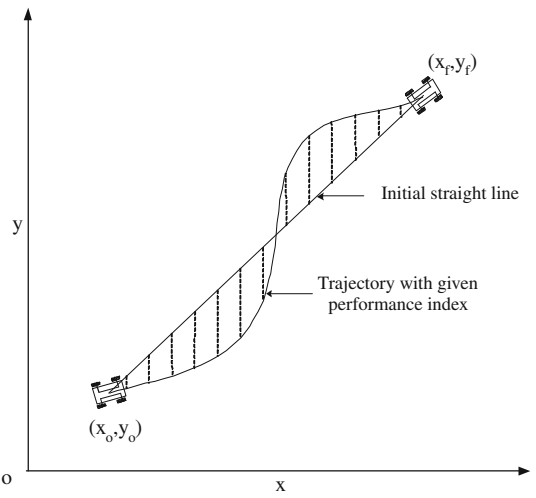


Fig. 4 The performance index and the initial straight line

- *Hard obstacle*: The obstacle should be avoided by the vehicle and the avoidance is mandatory, such as human beings, other vehicles and equipment in the environment.
- *Soft obstacle*: The obstacle is preferred to be avoided but can be run over/through by the vehicle, such as unpaved or gravel areas, potholes, and "splash" and "spray" on the wet road generated from other vehicle's wheels.

To be free of collision, the trajectory will be of the distance at least $(r_0 + r_i)$ from the ith obstacle. Using the relative velocity, the collision-avoidance criterion should only be considered for $x_{i'} \in [\underline{x}'_i, \overline{x}'_i]$, with $\underline{x}'_i = x_i^k - r_i - r_0$ and $\overline{x}'_i = x_i^k + r_i + r_0$. Assume the velocity of the obstacle is constant in each sampling period, the criterion for "hard" obstacle is

$$\left(y - y_i^k - v_{i,y}^k \tau\right)^2 + \left(x - x_i^k - v_{i,x}^k \tau\right)^2 \geq (r_i + r_0)^2, \tag{13}$$

where $\tau = t - t_k$ for $t \in [t_k, t_f]$.

Substituting Eq. 8 into Eq. 13 leads to

$$\min_{t \in [\underline{t}_i^*, \overline{t}_i^*]} G_i\left(t, \tau, a_6^k\right) \geq 0, \tag{14}$$

where

$$\begin{aligned}
\underline{t}_i^* &= t_0 + kT_s + \frac{x_i^k - x^k - r_i - r_0}{V_x^k - v_{i,x}^k}, \\
\overline{t}_i^* &= t_0 + kT_s + \frac{x_i^k - x^k + r_i + r_0}{V_x^k - v_{i,x}^k}, \\
\tau &= t - t_0 - kT_s, \\
G_i\left(t, \tau, a_3^k\right) &= g_2(x(t), k)\left(a_3^k\right)^2 + g_{1,i}(x(t), k, \tau) a_3^k + g_{0,i}(x(t), k, \tau).
\end{aligned}$$

$[\underline{t_i}^*, \overline{t}_i^*]$ is the time interval (if exists) during which a collision may happen, and

$$g_2(x(t),k) = \left[(x(t))^6 - X(B^k)^{-1}A^k\right]^2,$$
$$g_{1,i}(x(t),k,\tau) = 2\left[(x(t))^6 - X(B^k)^{-1}A^k\right] \cdot \left[X(B^k)^{-1}Y^k - y_i^k - v_{i,y}^k\tau\right],$$
$$g_{0,i}(x(t),k,\tau) = \left[X(B^k)^{-1}Y^k - y_i^k - v_{i,y}^k\tau\right]^2 + (x(t) - x_i^k - v_{i,x}^k\tau)^2 - (r_i + r_0)^2.$$

Clearly, analyzing the characteristic of G_i, we have

$$\Omega_{O,i} \triangleq \left\{ a_6^k : \min_{t\in[\underline{t}_i^*, \overline{t}_i^*]} G_i(t,\tau,a_6^k) \geq 0 \right\} = (-\infty, \underline{a}_{6,O,i}^k] \cup [\overline{a}_{6,O,i}^k, +\infty), \tag{15}$$

where

$$\underline{a}_{6,O,i}^k = \max_{x\in[x_0,x_f]} \frac{-g_{1,i} - \sqrt{(g_{1,i})^2 - 4g_2 g_{0,i,j}^l}}{2g_2},$$
$$\overline{a}_{6,O,i}^k = \min_{x\in[x_0,x_f]} \frac{-g_{1,i} + \sqrt{(g_{1,i})^2 - 4g_2 g_{0,i,j}^l}}{2g_2}. \tag{16}$$

See [16] for details regarding the case when $g_2 = 0$. Solution Eq. 16 is the collision-free interval of a_6^k for the ith obstacle in the environment. Since the "hard" obstacle must be avoided, the criterion for avoiding "hard" obstacles will be exactly the same as the collision-free criterion Eq. 15. Assume there are n_H "hard" obstacles in the sensor range. For each "hard" obstacle, in terms of adjustable parameter a_6^k, the collision-free interval is

$$\Omega_{H,l} \triangleq \Omega_{O,l}, \quad l = 1, 2, \ldots, n_H \tag{17}$$

Thus, the collision-free interval for all "hard" obstacles is

$$\Omega_H \triangleq \bigcap_{l=1}^{n_H} \Omega_{H,l}. \tag{18}$$

Choosing a_6^k from Ω_H and applying Lemma 0.1, for "hard" obstacles, we obtain the family of collision-free trajectories, the baseline for avoiding obstacles.

Since "hard" obstacles must be avoided, we assume the optimal performance index for avoiding all "hard" obstacles is J_H^*, and treat it as a baseline. The "soft" obstacles could be taken into account by relaxing the performance of this baseline, that is

$$J_S{}^* \leq \gamma J_H^*,$$

where γ is an adjustable relaxation parameter, reflecting the degree of desirability to avoid "soft" obstacles.

In the next section, by using the property of piecewise and parameterized trajectories, the collision-avoidance criterion for obstacles is mapped into collision-free intervals of the adjustable parameter, a_6^k, which gives the ability to find the optimal and real-time trajectory.

3 Optimal Solution

As described in Sect. 2, for representing the family of trajectories, the sixth-order polynomial is the simplest case, thus it is employed to present the framework of trajectory planning. In this section, collision-free trajectories are obtained and expressed by one adjustable parameter, a_6^k, mapped from collision-avoidance criterion. By combining with the performance index Eq. 12, an analytical and optimal feasible trajectory will be obtained.

3.1 Optimal Feasible Trajectory for "Hard" Obstacles

For "hard" obstacles, the analytical and optimal solution with respect to performance index Eq. 12 is stated in the following theorem

Theorem 0.1 *Consider the mobile vehicle given by Eq. 5 moving in an environment with dynamically moving "hard" obstacles, and the family of trajectories are parameterized by Eq. 8. Let*

$$a_6^{k*} = \frac{13(\frac{\partial^2 y}{\partial x^2}\big|_{x=x^k} + \frac{\partial^2 y}{\partial x^2}\big|_{x=x_f})}{12(x^k - x_f)^4} + \frac{117(\frac{\partial y}{\partial x}\big|_{x=x_f} - \frac{\partial y}{\partial x}\big|_{x=x^k})}{10(x^k - x_f)^5} + \frac{429\left[\frac{y_f - y_0}{x_f - x_0}(x^k - x_f) - (y^k - y_f)\right]}{10(x^k - x_f)^6}. \tag{19}$$

Then, under the boundary conditions Eq. 6, *for "hard" obstacles, the projection of* a_6^{k*} *on the set* Ω_H *generates an optimal collision-free trajectory while minimizing performance function* Eq. 12. *The definition of the projection is*

$$P_{\Omega_H}^{a_6^{k*}} \triangleq \{a_6^{k**} \in \Omega_H : \|a_6^{k**} - a_6^{k*}\| \leq \|a_6^k - a_6^{k*}\|, \quad \forall a_6^k \in \Omega_H\} \tag{20}$$

Proof It follows from Eq. 12 and 8 that

$$
\begin{aligned}
J_k(a_6{}^k) &= \int_{x^k}^{x_f} \left[X(B^k)^{-1}(Y^k - A^k a_6^k) + a_6^k x^6 - (y_f - y_0)(x - x_0)/(x_f - x_0) - y_0\right]^2 \mathrm{d}x \\
&= V_x^k \int_{t_0+kT_s}^{t_f} \left[(h_1)^2 (a_6^k)^2 + 2h_1 h_2 a_6^k + (h_2)^2\right] dt \\
&= f_1 (a_6^k)^2 + f_2 a_6^k + f_3,
\end{aligned} \tag{21}
$$

where

$$
\begin{aligned}
x(t) &= x^k + V_x^k (t - t_k), \\
h_1(t) &= (x(t))^6 - X(B^k)^{-1} A^k, \\
h_2(t) &= X(B^k)^{-1} Y - \frac{y_f - y_0}{x_f - x_0}(x(t) - x_0) - y_0,
\end{aligned} \tag{22}
$$

and

$$
\begin{aligned}
f_1 &= V_x^k \int_{t_0+kT_s}^{t_f} (h_1)^2 \mathrm{d}t, \quad f_2 = 2V_x{}^k \int_{t_0+kT_s}^{t_f} h_1 h_2 \mathrm{d}t, \\
f_3 &= V_x^k \int_{t_0+kT_s}^{t_f} (h_2)^2 \mathrm{d}t.
\end{aligned} \tag{23}
$$

Obviously, $f_1 > 0$ since $x^k \neq x_f$. Thus, $J_k(a_6^k)$ is a second order polynomial in terms of a_6^k, and its minimal value is achieved with the choice of

$$
a_6^{k*} = -\frac{f_2}{2f_1}. \tag{24}
$$

It follows from Eq. 23 that

$$
\begin{aligned}
f_1 &= -\frac{(x^k - x_f)^{12}(t_f - t_0)}{12012}, \\
f_2 &= \frac{(x^k - x_f)^6 (t_f - t_0)}{27720}\left[5\left(\frac{\partial^2 y}{\partial x^2}|_{x=x^k} + \frac{\partial^2 y}{\partial x^2}|_{x=x_f}\right)\right. \\
&\quad (x^k - x_f)^2 + 54\left(\frac{\partial y}{\partial x}|_{x=x_f} - \frac{\partial y}{\partial x}|_{x=x^k}\right)(x^k - x_f) \\
&\quad \left. +198\frac{(y_f - y_0)(x^k - x_f) - (y^k - y_f)(x_f - x_0)}{x_f - x_0}\right].
\end{aligned}
$$

Then, it is routine to obtain the analytical expression a_6^{k*} in Eq. 19. In the presence of "hard" obstacles, the objective is to find a_6^k that minimizes cost J_k subject to inequality constraint Eq. 14. $G_i(t, \tau, a_6^k)$ and $J_k(a_6^k)$ all are second order polynomials of a_6^k. This means $J_k(a_6^k)$ is the homeomorphism of a_6^k when $a_6^k \in (-\infty, a_6^{k*}]$ or $a_6^k \in [a_6^{k*}, +\infty)$. Analyzing this property, we know the projection of a_6^{k*} on the set Ω_H will generate the minimal J_k. This result means $P_{\Omega_H}^{a_6^{k*}}$ yields the optimal solution. Obviously, the optimal solution a_6^{k**} is an analytical solution. □

Remark 0.1 The "initial straight line" can be updated dynamically. In each sampling period, the beeline between point (x_k, y_k) and (x_f, y_f) is set to be "the initial straight line" for this period, that is, the performance index in terms of L_2 norm becomes

$$J_k(a_6^k) = \int_{x_k}^{x_f} \left[y - \frac{y_f - y_k}{x_f - x_k}(x(t) - x_k) - y_k \right]^2 dx.$$

Similarly, we can obtain an optimal trajectory by minimizing this performance index. Compared to the one generated by a_6^{k**}, it may be smoother and its length sometime turns shorter, but a general conclusion about which one is better cannot be drawn.

Remark 0.2 Instead of the "initial straight line", if the trajectory (x^*, y^*), generated by $a_6^k = a_6^{k*}$, is used as the reference trajectory, given the performance in the form of L_2 norm, that is,

$$J_k^*(a_6^k) = \int_{t_k}^{t_f} \left[(x - x^*)^2 + (y - y^*)^2 \right] \mathrm{d}t. \tag{25}$$

where

$$\begin{cases} x^*(t) & = & (F_k)^{-1} Y_k [1 \;\; t \;\; t^2 \;\; t^3], \\ y^*(x(t)) & = & (F_k)^{-1} (Y_k - E_k a_6^{k*}) [1 \;\; x \;\; x^2 \;\; x^3 \;\; x^4 \;\; x^5] + a_6^{k*} x^6, \end{cases}$$

the same solution a_6^{k**} as shown in Theorem 0.1 is achieved.

3.2 Optimal Feasible Trajectory for All Obstacles

In the environment, "soft" obstacles are expected to be avoided, nevertheless, if the cost for avoiding them, such as trajectory length and energy, is high, the vehicle may travel over them. Thus, the impact of avoiding "soft" obstacles should be taken into account. Such impact effect could be measured by relaxing the optimal performance index for avoiding "hard" obstacles with a given

tolerance level. The trajectory obtained in the previous part minimized the performance function while avoiding all "hard" obstacles, and it may run over "soft" obstacles. The impact of avoiding "soft" obstacles should be taken into account. Such impact effect could be measured by the corresponding degradation of performance index. Treated as a reference value the optimal performance index for avoiding "hard" obstacles, with a given tolerance level, the associated relaxation of performance index value is known, thus the "soft" obstacles are assigned different privilege to be shunned or run over. Suppose there are n_S "soft" obstacles in the sensor range, the corresponding relaxation inequality with respect to the performance index for each "soft" obstacles is

$$\Omega_{S1,j} \triangleq \left\{ a_6^k : \; J_k(a_6^k) < \gamma_j J_k(a_6^{k**}), \;\; j = 1, 2, \ldots, n_S \right\} \tag{26}$$

where $\gamma_j \geq 1$ refers to the relaxation factor for the jth "soft" obstacle, it is chosen according to application requirements. Its value reflects the desired level of trade-off at which the obstacle should be avoided. The bigger the value, the more the desired level, e.g., if $\gamma_j = +\infty$, the "soft" obstacle is actually a "hard" one.

The solution of Eq. 26 is

$$\Omega_{S1,j} = \left(\underline{a}_{6,S1,j}^k, \; \overline{a}_{6,S1,j}^k \right), \tag{27}$$

where

$$\begin{aligned} \underline{a}_{6,S1,j}^k &= \frac{-f_2 - \sqrt{(f_2)^2 - 4f_1\left(f_3 - \gamma_j J_k(a_6^{k**})\right)}}{2f_1}, \\ \overline{a}_{6,S1,j}^k &= \frac{-f_2 + \sqrt{(f_2)^2 - 4f_1\left(f_3 - \gamma_j J_k(a_6^{k**})\right)}}{2f_1}. \end{aligned} \tag{28}$$

This solution is the tolerance range of a_6^k for the vehicle to avoid the jth "soft" obstacle. Since the tolerance range is known, the criterion for avoiding "soft" obstacle is obtained by determining if "soft" obstacles are in the tolerance range. Therefore, collision-free criterion Eq. 13 is reapplied on "soft" obstacles, the collision-free interval for the jth "soft" obstacle is

$$\Omega_{S2,j} \triangleq \left\{ a_6^k : \min_{t \in [\underline{t}_j^*, \overline{t}_j^*]} G_j(t, \tau, a_6^k) \geq 0 \right\}. \tag{29}$$

Combing the tolerance range Eq. 27 with the collision-free interval Eq. 29, the interval of a_6^k for avoiding the jth "soft" obstacle is

$$\Omega_{S,j} = \overline{\Omega}_{S1,j} \bigcap \Omega_{S2,j}. \tag{30}$$

where $\overline{\Omega}_{S1,j}$ is the complement set of $\Omega_{S1,j}$.

The collision-avoidance interval for all "soft" obstacles is

$$\Omega_S \triangleq \bigcap_{j=1}^{n_S} \Omega_{S,j}. \tag{31}$$

Based on the results obtained from the previous parts, considering both kinds of obstacles, we obtain the optimal feasible trajectory in the following theorem.

Theorem 0.2 *Let the mobile vehicle given by* Eq. 5 *operate in an environment with dynamically-moving/static "hard" and "soft" obstacles, and the family of trajectories be parameterized by* Eq. 8. *Given* a_6^{k*} *defined in Eq.* 20, *under the boundary conditions Eq.* 6, *the projection of* a_6^{k*} *on the set* $\Omega \triangleq \Omega_H \bigcap \Omega_S$ *generates an optimal feasible trajectory for both kinds of obstacles. The definition of the projection is*:

$$P_\Omega^{a_6^{k*}} \triangleq \{a_6^{k***} \in \Omega : \|a_6^{k***} - a_6^{k*}\| \leq \|a_6^k - a_6^{k*}\|, \quad \forall a_6^k \in \Omega\} \tag{32}$$

Proof The steps to prove this theorem are the same as the ones in Theorem 0.1. □

Remark 0.3 Theoretically, envelopes of any obstacles are described by a combination of different size circles. Clearly, if we get the collision criteria for the circles forming the envelope of a certain obstacle, the criterion to avoid this obstacle is obtained by combining these criteria together. Thus, without losing generality, Theorems 0.1 and 0.2 can be extended to "hard" and "soft" obstacles with any kinds of shapes. The proof is similar to the one in Theorem 0.1.

3.3 Iterations for Improving the Performance

The basic idea of improving the performance is to constantly deform the feasible trajectory and reduce its performance index. For each iteration, intermediate points are chosen to divide the trajectory, obtained in previous iteration, into parts called primitive sub-trajectories. Then, each pair of neighboring points (including original boundary points and intermediate points) and their associated steering angles are treated as new boundary conditions, and the proposed method is used to generate new optimal feasible sub-trajectory between them. Finally, for each pair of neighboring points, the planner updates its corresponding sub-trajectory if the performance is improved. All these steps are repeated until a certain condition is satisfied.

Obviously, for setting such intermediate points exist numerous approaches. In the n_pth iteration, one method is to choose such points, whose coordinates on x-axis are $x_k + j(x_f - x_k)/n_p$, from the trajectory obtained in the previous interaction, where $j = 1, \ldots, n_p$. And, its iteration process is graphically shown in Fig. 5.

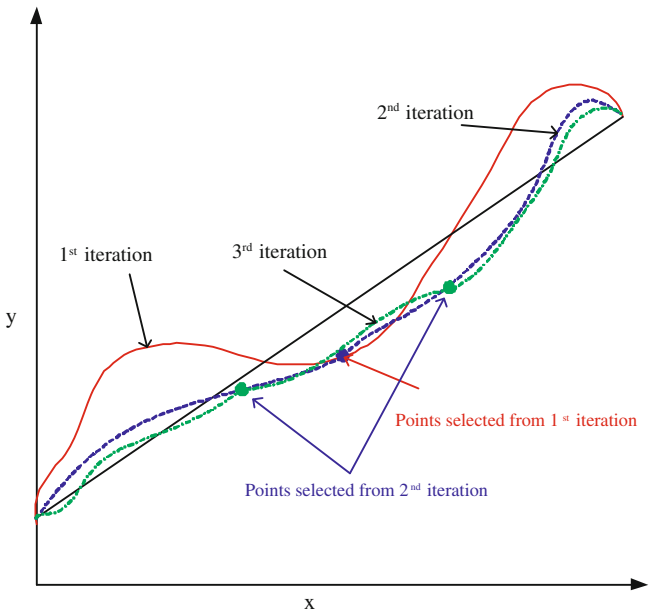


Fig. 5 Iterations to obtain the optimal solution

Since the performance index for each sub-trajectories is decreasing or keeping the same in each iteration, the performance of the total trajectory is improved gradually (at least keeps unchangeable).

4 Simulation

In this section, we apply our algorithm to two simple examples and show that the algorithm offers an effective method to generate a feasible trajectory in environments with "soft" and "hard" obstacles. The vehicle's settings are:

- Vehicle parameters: $r_0 = 1,\ l = 0.8$, and $\rho = 0.2$.
- Boundary conditions:

  Example 1: $(x_0, y_0, \theta_0, \phi_0) = (38, 36, -\frac{\pi}{4}, 0)$ and $(x_f, y_f, \theta_f, \phi_f) = (63, 32, -\frac{\pi}{4}, 0)$.
  Example 2: $(x_0, y_0, \theta_0, \phi_0) = (0, 0, \frac{\pi}{4}, 0)$ and $(x_f, y_f, \theta_f, \phi_f) = (17, 10, -\frac{\pi}{4}, 0)$, where $t_0 = 0$ and $t_f = 40$ s.

- Sensor range $R_s = 8$, implies that the vehicle has a limited sensor range so the vehicle detects the presence of obstacles intermittently; sampling period $T_s = 10$.

In the simulation examples, the scales are the same, obstacles are drawn every 5 s. And all quantities conform to a given unit system, for instance, meter, meter per second, etc.

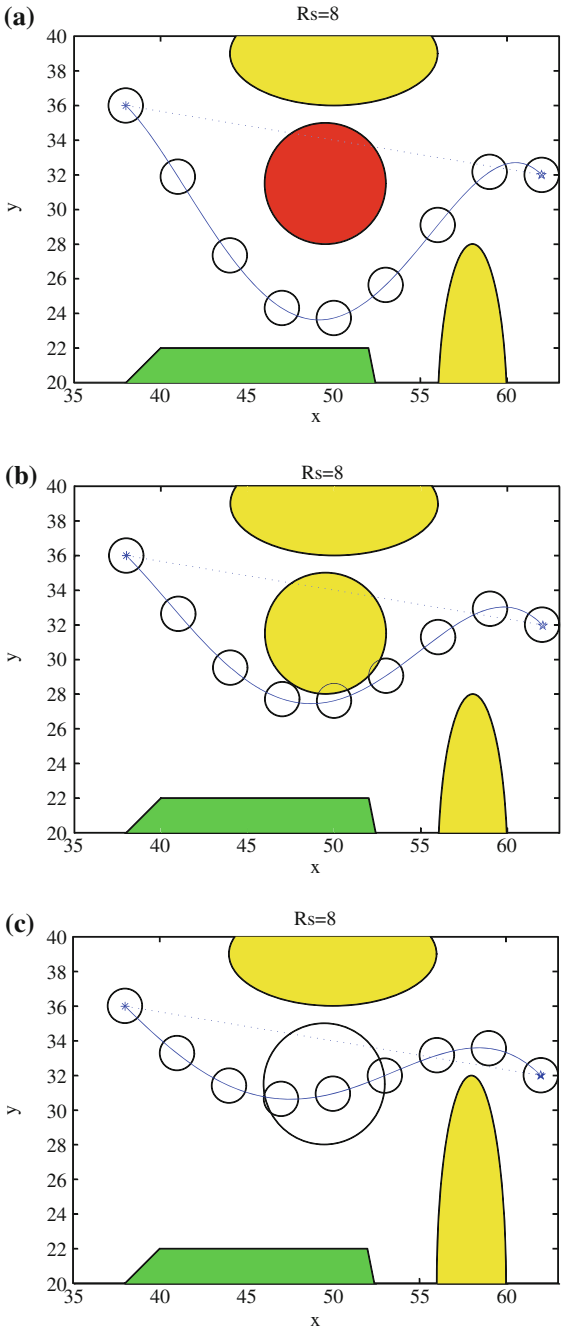


Fig. 6 Comparison of trajectories in a static environment: **a** the optimal trajectory avoiding "hard" static obstacles, **b** the optimal trajectory avoiding a "soft" obstacle with $\gamma = 3$, **c** The optimal trajectory avoiding a "soft" obstacle with $\gamma = 0$

Table 1 The settings of three-moving obstacles

	Initial locations of centroid	$t \in [0, 10]$	$t \in [10, 20]$	$t \in [20, 30]$	$t \in [30, 40]$
Obs 1	(5,2)	$v_1^1 = [0, 0.1]$	$v_1^2 = [0.5, 0.2]$	$v_1^3 = [0, -0.2]$	$v_1^4 = [0.1, -0.1]$
Obs 2	(6,4)	$v_2^1 = [-0.5, 0]$	$v_2^2 = [0.2, 0.1]$	$v_2^3 = [0.1, 0.1]$	$v_2^4 = [0.6, 0.1]$
Obs 3	(17,4)	$v_3^1 = [-0.2, -0.1]$	$v_3^2 = [-0.2, 0.1]$	$v_3^3 = [-0.1, 0.1]$	$v_3^4 = [-0.1, 0.1]$

Our first example is in the domain of static obstacles. Position of the vehicle is marked by a circle with radius 1 and positions of obstacles are marked by different size circles. In this example, three scenarios are studied with a rounded obstacle which is located at [50, 32] and of radius 1. The relaxation factors γ are different in these scenarios.

The simulation results are shown in Figs. 6a–c, respectively. In Fig. 6a, the rounded obstacle is a "hard" obstacle, the optimal a_6^k obtained by our approach is applied to generate the trajectory. The length of the trajectory is 34.2. This rounded obstacle is changed to "soft" obstacle in Fig. 6b and the relaxation factor $\gamma = 3$. For this obstacle, the vehicle tries to avoid it at the beginning, however, at last the vehicle chooses to traverse it according to the tolerance range. Thus the distance is shortened to 32.1. In Fig. 6, the relaxation factor γ for the "soft" obstacle is set to be 0, and the vehicle runs over it without considering it. We find that the trajectory is much closer to the initial straight line, and the trajectory length is reduced to 29.4.

To consider a representative example as a dynamical environment with moving obstacles, we present three cases, in which there exist three moving obstacles. The settings of the vehicle are the same as Example 1. The three obstacles' settings in these two scenarios are shown in Table 1.

The simulation results are shown in graphs of Fig. 7a, b, respectively. In Fig. 7a (graph), the vehicle uses the optimal trajectory to avoid three "hard" obstacles and the trajectory length is 23.9. For graph in Fig. 7b, obstacle 1 is changed to be a "soft" one with relaxation factor $\gamma = 3$. In the first two sampling periods, the vehicle also tries to run away from all obstacles, so it follows almost the same trajectory as shown in graph of Fig. 7a, however, at the beginning of the third-sampling period, it finds that the tolerance range will be violated if it shuns all of them, so it traverses the "soft" obstacle 1. Compared to graph of Fig. 7a, we find that the length of the trajectory in graph of Fig. 7b shrinks to 21.7. In graph of Fig. 7c, obstacle 1 is designed to be a complete "soft" one with relaxation factor $\gamma = 1$, thus, the vehicle will totally ignore obstacle 1 and the length of the path is 20.3.

5 Conclusion

A method for planning the trajectory of a vehicle in a dynamic environment with "hard" and "soft" obstacles has been developed. It is a parameterization approach, since the real-time and feasible trajectory is expressed by a fourth-order

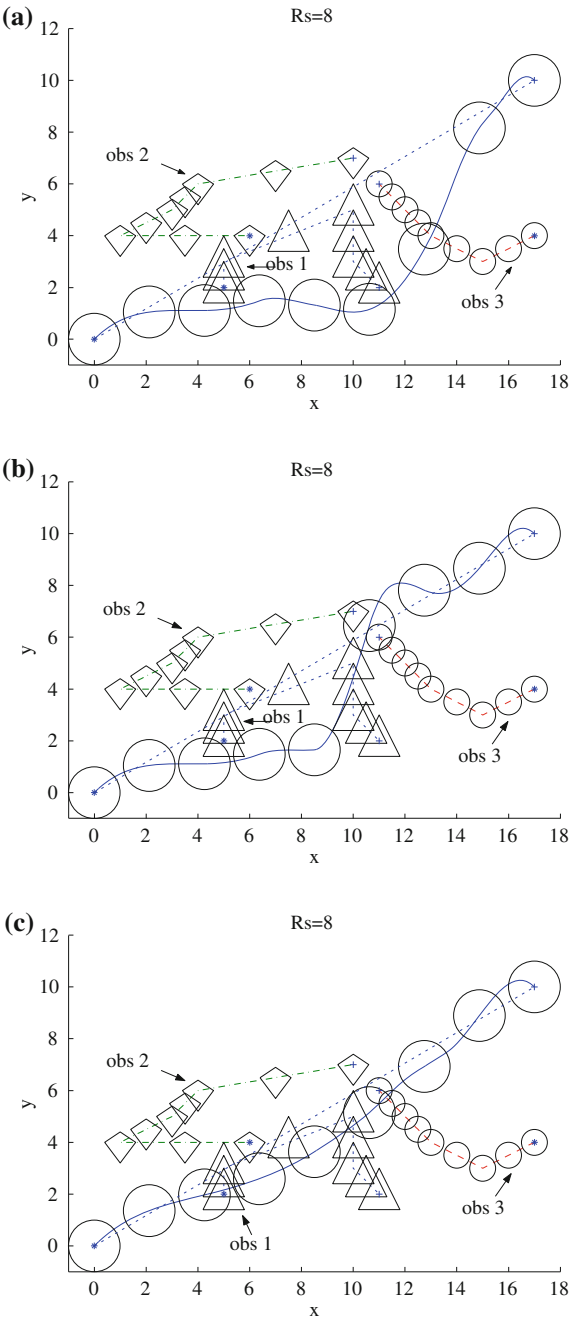


Fig. 7 Comparison of the trajectories in a dynamic environment: **a** the optimal trajectory avoiding three "hard" moving obstacles, **b** the optimal trajectory avoiding two "hard" moving obstacles and one "soft" moving obstacle with relaxation factor $\gamma = 3$, **c** the optimal trajectory avoiding two "hard" moving obstacles and one "soft" moving obstacle with relaxation factor $\gamma = 1$

parameterized polynomial. The only requirement to obtain the solution of an adjustable parameter makes it possible to generate the real-time trajectory. Key features of this approach are: (1) parameterized trajectories are employed to satisfy the boundary condition and kinematic equation; (2) collision-free trajectories are mapped into the associated intervals of the adjustable parameter; (3) an optimal trajectory, with performance index in the form of L_2 norm, for avoiding both kinds of obstacles is analytically obtained (Fig. 7).

This approach was illustrated in two examples showing the applicability of the proposed approach to the avoidance of "hard" and "soft" static/moving obstacles. Also demonstrated was the effectiveness of the optimal solution.

References

1. Khatib, O.: Real-time obstacle avoidance for manipulator and mobile robots. Int. J. Robot. Res. **5**, 90–98 (1986)
2. Charles, W. W.: A technique for autonomous underwater vehicle route planning. IEEE Trans. Oceanic. Eng. **3**, 199–204 (1990)
3. Hwang, Y.K., Ahuja, N.: A potential field approach to path planning. IEEE Trans. Robotics. Autom. **8**, 23–32 (1992)
4. Chuang, J.H.: Potential-based modeling of three-dimensional workspace for obstacle avoidance. IEEE Trans. Robotics. Autom. **14**, 778–785 (1998)
5. Ge, S.S., Cui, Y.J.: New potential functions for mobile robot path planning. IEEE Trans. Robotics. Autom. **16**, 615–620 (2000)
6. Sundar, S., Shiller, Z.: Optimal obstacle avoidance based on the Hamilton–Jacobi–Bellman equation. IEEE Trans. Robotics. Autom. **13**, 305–310 (2000)
7. Laumond, J-P., Jacobs, P.E., Taix, M., Murray, R.M.: A motion planner for nonholonomic mobile robots. IEEE Trans. Robotics. Autom. **10**, 577–593 (1994)
8. Reeds, J.A., Shepp, R.A.: Optimal paths for a car that goes both forward and backwards. Pacific. J. Math. **145**, 367–393 (1994)
9. Lazaro, J.L., Gardel, A.: daptive workspace modeling, using regression methods, and path planning to the alternative guide of mobile robots in environments with obstacles. Merging. Technol. Factory. Autom. 7th IEEE. International Conference, vol 1, pp. 529–534 (1999)
10. Eren, H., Chun, C.F., Evans, J.: Implementation of the spline method for mobile robot path control. Instrum. Meas. Technol. Conf. Proc. 16th IEEE. **2**, 739–744 (1999)
11. Piazzi, A., Lo Bianco, C.G., Bertozzi, M., Fascioli, A., Broggi, A.: Quintic G2-splines for the iterative steering of vision-based autonomous vehicles. Intell. Transp. Syst, IEEE Trans. **3**, 27–36 (2002)
12. Barraquand, J., Latombe, J.-C.: Nonholonomic multibody mobile robots: controllability and motion planning in the presence of obstacles. IEEE Int. Confer. Robotics. Autom. **1**, 2328–2335 (1991)
13. Divelbiss, A.W., Wen, J.T.: A path space approach to nonholonomic motion planning in the presence of obstacles. IEEE Trans. Robotics. Autom. **13**, 443–451 (1997)
14. Kant, K., Zucker, S.W.: Planning collision free trajectories in time-varying environments: a two level hierarchy. IEEE Int. Conf. Robotics. Autom. **1**, 1644–1649 (1988)
15. Erdmann, M., Lozano-Perez, T.: On multiple moving objects. IEEE Int. Conf. Robotics. Autom. 1419–1424 (1986)
16. Qu, Z., Wang, J., Plaisted, C.E.: A new analytical solution to mobile robot trajectory generation in the presence of moving obstacles. IEEE Trans. Robotics. **20**, 978–993 (2004)

17. Yang, J., Daoui, A., Qu, Z., Wang, J., Hull, R.A.: An optimal and real-time solution to parameterized mobile robot trajectories in the presence of moving obstacles. IEEE. Int. Conf. Robotics. Autom. **1**, 4412–4417 (2005)
18. Yang, J., Qu, Z., Hull, R.A.: Trajectory planning for UGVs in an environment with "Hard" and "Soft" obstacles. AIAA GNC. Conf. **1**, 6105 (2006)
19. Tiibury, D., Murray, R.M., Sastsy, S.S.: Trajectory generation for the N-trailer problem using goursat normal form. Autom. Control IEEE Trans. **40**, 802–819 (1995)

Planning Continuous Curvature Paths Using Constructive Polylines

Joshua Henrie and Doran Wilde

Abstract Previous methods for planning clothoid-based continuous curvature paths aim at minimizing path length. However, minimal length paths are not always smooth, natural, and drivable. A new method of generating clothoid-based trajectories is discussed using constructive polylines. The goal of the motion planner is to create a path for a large autonomous car-like vehicle in human driving environments. Thus, the trajectories generated by the motion planner must be smooth, drivable, and natural such that the vehicle can follow the planned path on human roadways.

1 Introduction

The purpose of motion planning is to provide a path that autonomous car-like vehicles can physically drive. Motion planning has been defined as the study of open loop controls that steer a mobile robot from a specified initial state to a specified final state over a certain time interval [9]. Motion planning can be done by constructing trajectories that have steering controls (curvature and sharpness) as well as velocity controls (velocity and acceleration/deceleration) embedded in the data structure of the trajectory. The motion planner described in this chapter constructs smooth, natural, drivable trajectories for a large car-like vehicle. The planner generates these trajectories using constructive polylines to create a path

J. Henrie (✉)
Raytheon Missile Systems, 1151 E. Hermans Road, Tucson, AZ 85734, USA
e-mail: jhenrie2@gmail.com

D. Wilde
Department of Electronic and Computer Engineering, Brigham Young University, 459 Clyde Building Provo, UT 84602, USA
e-mail: wilde@ee.byu.edu

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_7, © Springer-Verlag London Limited 2012

composed of a combination of straight line segments, clothoid segments, and circular arcs. A clothoid is a curve whose curvature increases/decreases linearly with the distance traveled on the arc. Curvature is the change of heading angle (in radians) per unit distance traveled on the arc (in meters), and is also the inverse of the turning radius. Positive curvature means the vehicle is turning left, and negative curvature means the vehicle is turning right. Sharpness is the rate of change of curvature with respect to distance traveled. Straight line segments and circular arcs have zero sharpness, and clothoid segments have constant non-zero sharpness. Constructive polylines are straight lines used to specify the approximate desired path.

1.1 Previous Work

There are two commonly used motion planning methods. The first is to pre-compute a set of feasible continuous curvature paths and then heuristically choose the best path that arrives at the desired destination and heading, used by Coombs [1], Knowles [8], and Piavtoraiko and Kelly [10]. The second is to calculate a continuous curvature trajectory that traverses the distance between two points and ends with the desired heading. Both Shin and Singh [12], and Fraichard and Scheuer [4] use this second method . This chapter uses the second method.

Pre-computing a set of feasible paths and heuristically choosing the best path works well for motion planners integrated with path planners. Another type of path planner outputs waypoints (points containing an *xy* position and a heading) to specify approximate paths that avoid objects and move the vehicle toward its desired destination. The path planner plans an approximate route specified by waypoints and then passes the waypoints to the motion planner where the exact trajectories are generated. A trajectory is a path, or more precisely a set of driving instructions that an autonomous vehicle follows to reach the desired destination.

Shin and Singh [12] use postures (a point and the heading and curvature at that point) to determine the trajectories that connect the postures. The motion planner described in this chapter uses trigonometry to compute the trajectories that connect a set of zero curvature points with specified initial and final headings. Fraichard and Scheuer [4] use a geometry of lines and circles to calculate trajectories that approximate Reeds and Shepp [11] optimal length paths.

Paths called “optimal paths” in the previous literature refer to minimal length paths. Dubins showed that a length-optimal discontinuous curvature path connecting an initial state and a final state of a mobile robot consists of only straight line segments and circular arc segments of maximal allowable curvature [3]. Reeds and Shepp [11] built upon Dubins’ work and gave a complete characterization of the shortest possible discontinuous curvature paths. They also extended Dubins’ work by adding cusps (reversals in travel direction) to their paths, and enumerated the possible types of optimal paths showing that an optimal path required no more than five segments and no more than two direction reversals. Fraichard and Scheuer [4]

showed that under the continuous curvature assumption, near optimal paths consist of straight lines, circular arc segments of maximal allowable curvature, and clothoid curves. These use clothoid curves that have maximum allowable sharpness to connect straight-line segments with the circular arc segments of maximal curvature. Remember that sharpness is change of curvature with respect to distance traveled and is also linearly related to the amount of steering motion along the trajectory [12] (i.e., related to the rate at which the steering wheel of the autonomous car-like vehicle turns). Fraichard and Scheuer [4] paths are near optimal and within 10% of the length of Reeds and Shepp paths.

1.2 Motivation

Motivation for this motion planner is to generate a smooth, natural, drivable trajectory between two waypoints in forward motion similar to the way a human drives. Humans do not drive paths that have discontinuous curvature or only turn at maximum allowable curvature and sharpness like those generated by the prior work discussed above. For example, if a human were to drive paths that always made turns at maximum sharpness and curvature, the driving experience would be akin to that of driving with a teenager who tries to turn as sharply as possible at every turn. Such a driving experience would be uncomfortable and probably scary. The motion planner presented in this chapter uses the smallest curvature and sharpness required to obtain a trajectory between two waypoints. In other words, the trajectory planned by the motion planner turns the steering wheel of a car-like vehicle as little and as slowly as possible.

Once a path has been generated by the motion planner, the maximum safe velocity for the trajectory is calculated, and hence, the travel time is minimized along the trajectory. Since time-optimal trajectories do not necessarily require that car-like robots or vehicles perform turns at maximal sharpness and curvature, we have been investigating methods for constructing continuous curvature trajectories where the curvature and sharpness are left as variables. Therefore, instead of constructing paths/trajectories from maximal curvature circular arcs, maximal sharpness clothoids, and line segments, we investigate a new method for computing trajectories based on constructive polylines that allows curvature and sharpness of the trajectory to be computed as a function of the distance traveled and the desired change of heading.

Before proceeding further, the terms smooth, drivable, and natural need to be defined. Smooth means the curvature of the trajectory is continuous. Drivable means the trajectories are acceptable for driving on human roadways. For example, while left u-turns are acceptable on most roadways, right u-turns are not. The last term, natural, is more difficult to define, but essentially it means that motion planned paths are similar to paths that a human would drive under normal driving conditions (e.g., little or no traffic, no icy/rainy conditions, etc.). While driving in these conditions, humans do not continually turn the steering wheel as quickly and

sharply as possible to make a turn. This behavior is unnatural for humans, as they tend to turn the steering wheel as little as possible, and when they do turn, they turn only as fast and sharply as is required to make the turn. While human behavior is difficult to quantize, the motion planner described in this chapter uses the smallest sharpness and curvature required to generate a path with the desired ending point and heading.

From these definitions of smooth, drivable, and natural, Dubins, and also Reeds and Shepp, paths are not smooth, drivable, or natural, as they do not have continuous curvature and require every turn to have maximum sharpness [3, 11]. Fraichard and Scheuer paths are smooth and drivable, but not natural, as they require every turn to have maximum sharpness and curvature.

1.3 Overview of the Motion Planner

Clothoid-based trajectories are generated by the motion planner using constructive polylines. A constructive polyline connects the points where the curvature of the trajectory is zero, which includes straight lines and inflection points (Fig. 1). These construction polylines are defined and used differently from the control lines used by Walton and Meek to draw clothoid arcs in a graphic editor [13].

Section 2 describes an important primitive method, called Clothoid_Turn(), used for motion planning trajectories described by constructive polylines, which has been proposed. The Clothoid_Turn() algorithm generates a curve by using only two values to calculate the arc length, L, the final curvature, κ, and the sharpness, α of the curve. The two values are: (1) the length d of the polyline segment and (2) the heading deflection, δ, which is also the angle at which the curve is incident to the polyline.

Figure 2 shows two types of curves that can be generated by Clothoid_Turn(). Figure 2a, b show the curves in the xy domain, and Fig. 2c, d show the curvature diagrams of Fig. 2a, b, respectively. A curvature diagram is convenient to work with because it visually relates the curvature (vertical axis), sharpness (slope), arc length (horizontal axis), and change of heading (area under the curve) of a clothoid.

Clothoid_Turn() generates a simple clothoid (Fig. 2a, c) if the final curvature is less than or equal to the maximum curvature the vehicle can drive, $\kappa_{\max}$. Otherwise, a combination clothoid-arc curve is generated (Fig. 2b, d) such that the curve does not exceed the $\kappa_{\max}$ of the vehicle. (Remember that $\kappa_{\max}$ is the inverse of the minimum turning radius of the vehicle.)

The Clothoid_Turn() method only really plans half a curve. If the curve is symmetric (Fig. 3a, b), then a single call to the clothoid turn method plans both halves of the curve, and $d = \frac{\text{Length(polyline)}}{2}$. A curve of two clothoids that begins and ends with zero curvature is called a *clothoid set*.

Section 3 then describes how arcs are inserted into a clothoid set using Newton's root approximation method to find the angle and length of an arc. Section 4

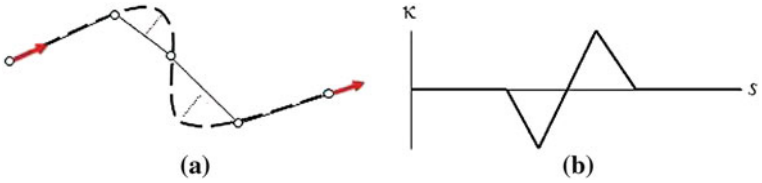


Fig. 1 **a** A trajectory (*dash-dotted line*) and the associated constructive polyline (*solid lines*). **b** The curvature plot for the trajectory on the *left*

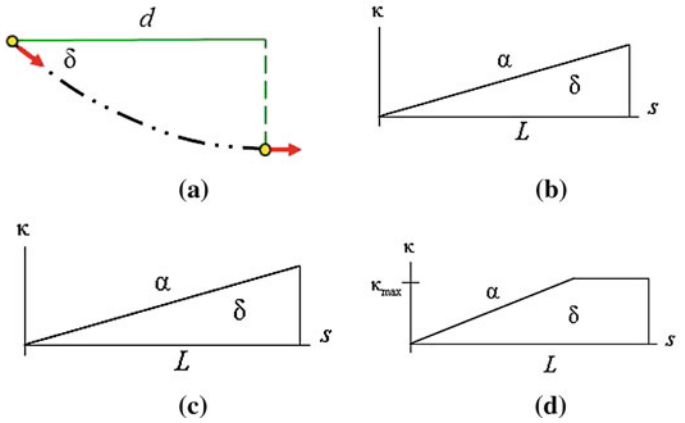


Fig. 2 **a** Clothoid segment given heading a deflection δ and polyline length d. **b** Clothoid-arc segment given heading deflection δ and polyline length d. **c** Curvature diagram for clothoid segment with arc-length L, final curvature κ, sharpness (*slope*) α, and deflection δ (*area under the curve*). **d** Curvature diagram for clothoid-arc segment with arc-length L, final arc curvature κ, sharpness α, and deflection δ

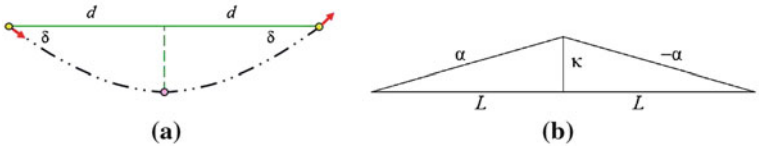


Fig. 3 **a** Symmetric curve. **b** Symmetric curvature diagram

uses the results of Sects. 2 and 3, in conjunction with polylines, to create trajectories from two points and their corresponding headings, similar to Fig. 1a.

2 Clothoid Equations and Planning a Basic Clothoid Curve

This section describes how a clothoid can be scaled without altering its shape and describes an algorithm which calculates a rotated and scaled clothoid with curvature, κ, sharpness, α, and length, L, given a constructive polyline length, d, and change of heading, δ.

First, a change of variables is performed on the clothoid equation, defined by Eq. 1, from two unknowns, clothoid sharpness, α and clothoid length, s, to one known and one unknown, clothoid change of heading, δ, and clothoid curvature, κ, respectively. Then the clothoid equation is put into the terms of the xy plane because the constructive polyline, change of heading, and waypoints are in the xy plane. Finally, the equation is scaled and rotated to create a trajectory of the correct length and orientation.

2.1 Converting the Clothoid Equation

A clothoid is a curve with continuous increasing/decreasing linear curvature and is defined in the complex plane by Eq. 1 derived by Kimia et al. [7].

$$F(s) = C_0 + \text{sign}(\alpha)\sqrt{\frac{\pi}{|\alpha|}}e^{i\left(\theta_0 - \frac{\kappa_0^2}{2\alpha}\right)}\left[C\left(\frac{\kappa_0 + \alpha s}{\sqrt{\pi|\alpha|}}\right) - C\left(\frac{\kappa_0}{\sqrt{\pi|\alpha|}}\right)\right] + i\sqrt{\frac{\pi}{|\alpha|}}e^{i\left(\theta_0 - \frac{\kappa_0^2}{2\alpha}\right)}\left[S\left(\frac{\kappa_0 + \alpha s}{\sqrt{\pi|\alpha|}}\right) - S\left(\frac{\kappa_0}{\sqrt{\pi|\alpha|}}\right)\right], \quad \alpha \neq 0. \tag{1}$$

C_0 is the starting point of the clothoid; θ_0 the initial clothoid heading; κ_0 the initial clothoid curvature; α is the clothoid sharpness; s the distance along the clothoid, and the Fresnel sine and cosine are defined as

$$C(s) = \int_0^s \cos\left(\frac{\pi}{2}u^2\right)\mathrm{d}u \text{ and } S(s) = \int_0^s \sin\left(\frac{\pi}{2}u^2\right)\mathrm{d}u.$$

The term $\kappa_0 + \alpha\, s$ is the instantaneous curvature after traveling a distance s along the trajectory.

While Eq. 1 is valid for all clothoids except where $\alpha = 0$, which is the case for straight lines and circular arcs, it is in terms of two unknown variables α and s. To reduce the number of unknowns, the clothoid equation is converted from the two

unknown variables into one unknown variable, κ, and one known variable, δ. The variable κ was chosen to be the unknown variable in the equation, and not α or s, because of the clothoid scaling process, which is described in Sect. 3.

Converting Eq. 1 can be simplified by assuming that the initial and final points of a clothoid set begin and end with zero curvature, $\kappa_0 = 0$ (Fig. 3b). In essence, this means that the vehicle's wheels must be straight upon entering or exiting a trajectory. Assuming $\kappa_0 = 0$ greatly simplifies the process of creating continuous curvature trajectories by allowing the catenation of arbitrary clothoid sets and straight-line segments. If the vehicle exits a trajectory with non-zero curvature, an additional clothoid must be added to create a continuous curvature trajectory that can connect to a straight line. However, if the vehicle exits a trajectory with zero curvature then it may enter a tangential straight-line segment without needing to add additional clothoids for the transition.

Without loss of generality, a trajectory can be computed for the normal case $\theta_0 = 0$ and $C_0 = 0$, and then rotated and translated into its correct orientation. Section 3 further explains the process of rotating and translating a clothoid curve. After applying the assumptions of $\kappa_0 = 0$, $\theta_0 = 0$ and $C_0 = 0$, Eq. 1 reduces to Eq. 2.

$$F(s) = \sqrt{\frac{\pi}{|\alpha|}}\left[\text{sign}(\alpha)C\left(\frac{\alpha s}{\sqrt{\pi|\alpha|}}\right) + iS\left(\frac{\alpha s}{\sqrt{\pi|\alpha|}}\right)\right], \quad \alpha \neq 0. \tag{2}$$

Although Eq. 3 has been significantly reduced from Eq. 1, it is stated only in terms of the unknown curve length, s, and sharpness, α. The inputs of the motion planner consist of only the initial and final points and their associated headings, θ_0 and θ_f. Therefore, the length, curvature, and sharpness of the desired clothoid for the trajectory are not inputs. The change in heading ($\delta = \theta_f - \theta_0$), also known as deflection [4], is easily computed from the initial and final points. To derive the primitive clothoid turn method referred to in the introduction, Eq. 3 can be rewritten as Eq. 8, in terms of δ and κ instead of s and α. The derivation for Eq. 8 is given in the discussion below. The purpose of this change of variables is to use δ and a constructive polyline to scale a clothoid linearly with κ. The process of determining the value of κ and scaling a clothoid is discussed in Sect. 3.

As stated previously, a curvature diagram showing the curvature, κ, with respect to the distance traveled along the curve, s (Fig. 4a) is a convenient diagram to work with because it visually relates the curvature (vertical axis), sharpness (slope), length (horizontal axis), and change of heading (area under the curve) of a clothoid. Figure 4 shows a clothoid in the s-κ domain and the xy domain. From this figure, the relationships between variables δ and κ and variables s and α are visible (3).

$$\kappa = \alpha s, \quad \delta = \frac{\alpha s^2}{2}, \quad \alpha = \frac{2\delta}{\kappa^2}. \tag{3}$$

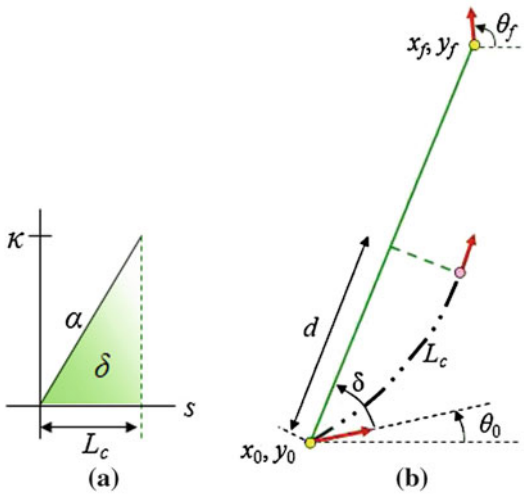


Fig. 4 A clothoid in the **a** s-κ domain and **b** xy domain

Using Eq. 3, the inner terms of the Fresnel integrals in Eq. 2 are put in terms of δ.

$$\frac{\alpha s}{\sqrt{\pi|\alpha|}} = \left\{ \begin{array}{ll} \alpha > 0 & : \sqrt{\frac{\alpha^2 s^2}{\pi\alpha}} = \sqrt{\frac{\alpha s^2}{\pi}} = \sqrt{\frac{2\delta}{\pi}} \\ \alpha < 0 & : (-1)\sqrt{\frac{(-\alpha)^2 s^2}{\pi(-\alpha)}} = (-1)\sqrt{\frac{(-\alpha)s^2}{\pi}} = (-1)\sqrt{\frac{2(-\delta)}{\pi}} \end{array} \right\}$$
$$= \frac{\delta}{|\delta|}\sqrt{\frac{2|\delta|}{\pi}} = \text{sign}(\delta)\sqrt{\frac{2|\delta|}{\pi}} \tag{4}$$

where

$$\text{sign}(\alpha) = \frac{\alpha}{|\alpha|} = \frac{\kappa}{|\kappa|} = \frac{\delta}{|\delta|} = \text{sign}(\delta). \tag{5}$$

For a single clothoid starting with zero curvature, Eq. 5 can be derived from Fig. 4.

$$\sqrt{\frac{\pi}{|\alpha|}} = \left\{ \begin{array}{ll} \alpha > 0 & : \sqrt{\frac{\pi}{\alpha}} = \frac{\sqrt{2\pi\delta}}{\kappa} \\ \alpha < 0 & : \sqrt{\frac{\pi}{-\alpha}} = \frac{\sqrt{2\pi(-\delta)}}{-\kappa} \end{array} \right\} = \sqrt{\frac{\delta}{|\alpha|}} = \frac{\sqrt{2\pi|\delta|}}{|\kappa|}. \tag{6}$$

Next, a change of variables is performed on the constant term in front of the Fresnel equations in Eq. 2 in terms of δ and κ by using the same Eq. 3. Substituting Eqs. 4–5 into Eq. 2, the final transformed clothoid equation is

$$F\left(\frac{2\delta}{\kappa}\right) = \frac{\sqrt{2\pi|\delta|}}{|\kappa|}\left[C\left(\sqrt{\frac{2|\delta|}{\pi}}\right) + \text{sign}(\delta) i\, S\left(\sqrt{\frac{2|\delta|}{\pi}}\right)\right]. \tag{7}$$

2.2 Translating the Clothoid Equation into the xy Plane

As the constructive polylines, the change of heading, and the waypoints are in the *xy* plane and the purpose of the constructive polylines is to scale clothoids in the *xy* plane, it is advantageous to translate Eq. 7 into the terms of the *xy* plane. Translating the equation into the *xy* plane is also the foundation for the clothoid scaling and rotating process discussed in Sect. 3. To translate let

$$F\left(\frac{2\delta}{\kappa}\right) = x\left(\frac{2\delta}{\kappa}\right) + i\, y\left(\frac{2\delta}{\kappa}\right) \tag{8}$$

where

$$\begin{aligned} x\left(\frac{2\delta}{\kappa}\right) &= \frac{\sqrt{2\pi|\delta|}}{|\kappa|} C\left(\sqrt{\frac{2|\delta|}{\pi}}\right) \text{and} \\ y\left(\frac{2\delta}{\kappa}\right) &= \text{sign}(\delta)\frac{\sqrt{2\pi|\delta|}}{|\kappa|} S\left(\sqrt{\frac{2|\delta|}{\pi}}\right). \end{aligned} \tag{9}$$

An important and key observation from Eq. 2 is that κ determines the size of the clothoid and δ determines the shape of a clothoid. Therefore, κ can be used to scale the size of a clothoid without altering its shape as shown in Fig. 5.

$$x\left(\frac{2\delta}{\kappa}\right) = \frac{1}{|\kappa|}x(2\delta) \quad \text{and} \quad y\left(\frac{2\delta}{\kappa}\right) = \frac{1}{|\kappa|}y(2\delta) \tag{10}$$

where the shape of the clothoid is determined by δ alone:

$$x(2\delta) = \sqrt{2\pi|\delta|}\; C\left(\sqrt{\frac{2|\delta|}{\pi}}\right) \quad \text{and} \quad y(2\delta) = \text{sign}(\delta)\sqrt{2\pi|\delta|}\; S\left(\sqrt{\frac{2|\delta|}{\pi}}\right). \tag{11}$$

Figure 5a,b shows *similar clothoids*. Figure 5a is a clothoid produced from Eq. 11 given only the deflection angle δ. Figure 5b is a scaled replica of that clothoid produced from Eq. 2. The scale factor $\frac{1}{|\kappa|}$ is only a function of the curvature κ. Since these two clothoids differ only in their scale, their geometry is *similar*. They share the same initial and final headings and deflection angle. However, distances *x,y*, d_2, and the arc length of the curve are all scaled linearly by the factor $\frac{1}{|\kappa|}$.

2.3 Rotating and Scaling Clothoids

The objective here is to find an algorithm to compute the clothoid curve that deflects by an angle δ while traveling forward a distance d as shown in Fig. 2.

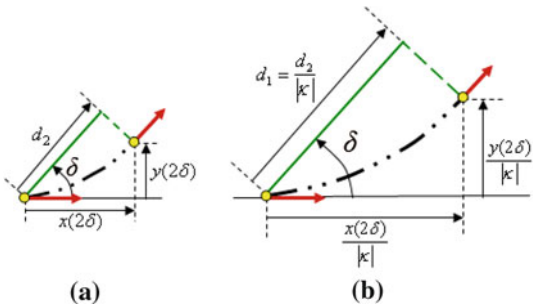


Fig. 5 Clothoid scaling. Clothoids in the xy plane **a** when $\kappa = 1$, **b** scaled by $\frac{1}{|\kappa|}$

The resulting clothoid is described using three parameters, arc length L, curvature κ, and sharpness α. Thus, given inputs δ and d, compute L, κ, and α.

Given the deflection angle, δ, the shape of the clothoid is fixed by Eq. 11 as shown in Fig. 5a. That clothoid is then rotated by $-\delta$ so it is in the normal position as illustrated in Fig. 2a. After being rotated, the curve is then scaled to the correct size as measured by the forward distance traveled (half the length of the constructive polyline).

2.3.1 Rotating and Scaling Algorithm

To scale and rotate a clothoid first let $\kappa = 1$ and compute $x(2\delta)$ and $y(2\delta)$, which are functions of only δ. Next, rotate $x(2\delta)$ and $y(2\delta)$ by $-\delta$; the x-component of the resulting point is d_2 in Eq. 12.

$$d_2 = \cos(\delta)x(2\delta) + \sin(\delta)y(2\delta). \tag{12}$$

As described in the next section, the distance d_1 in Fig. 5b is computed by Eq. 15. After computing both d_1 and d_2 the curvature, κ, can be computed from Eq. 13.

$$\kappa = \text{sign}(\delta)\frac{d_2}{d_1}. \tag{13}$$

Once κ is computed, the length and sharpness of the scaled clothoid (Fig. 5b) can be found using Eq. 14. The result is a clothoid turn with the desired length and change of heading. The clothoid turn algorithm with inputs d_1 and δ and outputs κ, α and L can be seen in Fig. 6.

$$\alpha = \frac{\kappa^2}{2\delta}, \quad L = \frac{\kappa}{\alpha}. \tag{14}$$

The Clothoid_Turn() algorithm uses the property of clothoids that states that they can be scaled while retaining their shape and deflection to produce clothoids based on the length of a polyline and the desired change of heading. Using the algorithm, basic trajectories can be created from two points and their associated headings.

Clothoid_Turn(d_1, δ)

{

if $\delta == 0$ //Straight Line

{

$\kappa = 0$

$\alpha = 0$

$L = d_1$

}

else if $\delta > 0$ //Left Turning Clothoid

{

$$d_2 = \sqrt{2\pi|\delta|}\left[\cos(\delta)C\left(\sqrt{\frac{2|\delta|}{\pi}}\right) + \sin(\delta)S\left(\sqrt{\frac{2|\delta|}{\pi}}\right)\right]$$

$$\kappa = \frac{d_2}{d_1}$$

$$\alpha = \frac{\kappa^2}{2\delta}$$

$$L = \frac{\kappa}{\alpha}$$

}

else if $\delta < 0$ //Right Turning Clothoid

{

$$d_2 = \sqrt{2\pi|\delta|}\left[\cos(\delta)C\left(\sqrt{\frac{2|\delta|}{\pi}}\right) - \sin(\delta)S\left(\sqrt{\frac{2|\delta|}{\pi}}\right)\right]$$

$$\kappa = -\frac{d_2}{d_1}$$

$$\alpha = \frac{\kappa^2}{2\delta}$$

$$L = \frac{\kappa}{\alpha}$$

}

}

Fig. 6 Clothoid_Turn() algorithm. Outputs of the algorithm are clothoid curvature, κ, sharpness, α, and length, L

3 Constructing Trajectories Using Clothoids, Arcs and Lines

Trajectories can be planned between two points and the initial heading using polylines and the clothoid scaling process described by the Clothoid_Turn() algorithm. Because the motion planner is used for large car-like vehicles which have a large minimum turning radius (turning radius is the inverse of curvature), there is a maximum curvature (max curvature $= \frac{1}{\text{min radius}}$) that can be driven. If a cloithoid curve reaches this maximum curvature, then the steering wheel cannot be

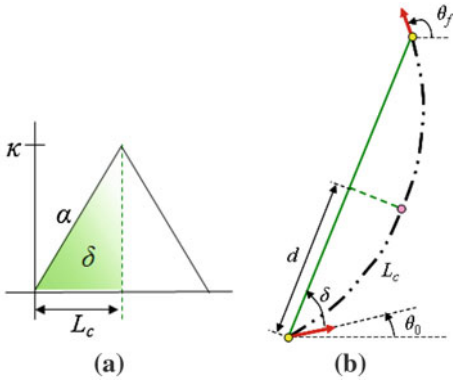


Fig. 7 Symmetric clothoid set with unconstrained curvature in the **a** $s\kappa$ plane and **b** xy plane

turned any more. Clothoid curves merge into circular arcs when maximum curvature is reached. Therefore, trajectories for large car-like vehicles consist of circular arc segments, straight line segments, and clothoid segments. The clothoids are used to make smooth continuous curvature transitions between the straight line segments and circular arc segments.

3.1 Constructing Trajectories

Remember that trajectories begin and end with zero curvature. Therefore, the simplest trajectory consists of a single straight line. Following the straight line, the next simplest trajectory consists of two clothoids of the same sharpness, the second clothoid mirroring the first (Fig. 7). Two clothoids with the same magnitude of sharpness but opposite signs are called a symmetric clothoid set.

If the change of heading of the clothoid set is positive then the trajectory is a left turn, and if the change of heading of the set is negative then the trajectory is a right turn. To create a symmetric clothoid set choose a constructive polyline with length d, where

$$d = \frac{\sqrt{(x_f - x_0)^2 + (y_f - y_0)^2}}{2}. \tag{15}$$

A constructive polyline with length d creates a clothoid turn that covers half the distance between the initial and final points. Mirroring the clothoid covers the second half of the distance and a trajectory is created (Fig. 7). It should be noted that choosing d to be half the distance between the initial and final points is not required, but it is a simple and effective solution.

A clothoid set is a simple trajectory because it is based on a single construction polyline and begins and ends with zero curvature. At this point, the definition of a clothoid set is expanded to include an optional arc which is an upper bound on

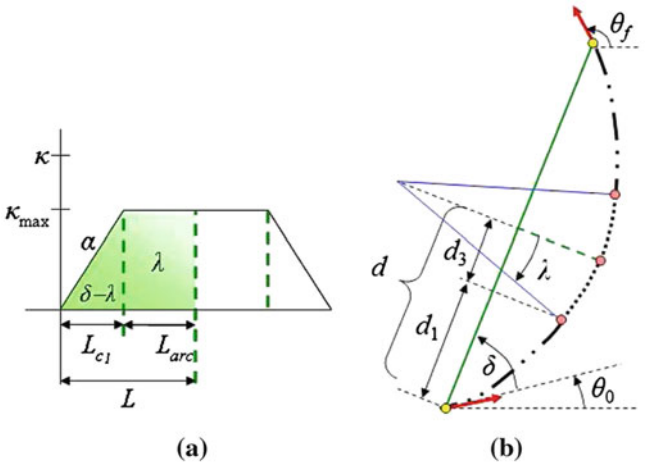


Fig. 8 Clothoid set with an inserted symmetric arc to constrain curvature in the **a** $s\kappa$ plane and **b** xy plane. Note that d_1 is less than the d in Eq. 15, and d_3 makes up the remaining distance to the halfway point between the initial and final points of the desired trajectory. L_c is the length of the clothoid and L_{arc} is the length of the arc. The angle of the arc, λ (i.e., the change of heading of the arc), is a part of the change of heading of half the trajectory

curvature. Adding an arc to a clothoid set creates the same type of trajectory as in Fig. 7, but with an upper-bounded curvature (Fig. 8). This extended definition of a clothoid set is similar to a "CC-Turn" by Fraichard and Scheuer [4], except that it is not required to have maximum sharpness and curvature.

3.2 Creating a Clothoid-Arc Set

With the exception of a few specialty vehicles, all car-like vehicles have a minimum turning radius. It is possible that the clothoid computed to turn a given deflection over a given forward distance might have a maximum curvature that exceeds the maximum curvature, $\kappa_{\max}$, that the vehicle can drive. When this physical limitation is exceeded, the clothoid segment must be replaced with clothoid and circular arc combination segment that respect the $\kappa_{\max}$ limitation. In other words, while making a turn the steering wheel of a vehicle eventually stops and the vehicle cannot turn any sharper (minimum turning radius). When the vehicle cannot turn any sharper it leaves a clothoid curve and enters a circular arc. Thus, an arc needs to be introduced into a trajectory to ensure that the curvature of the trajectory does not exceed the vehicle's maximum curvature (i.e., minimum turning radius). Figure 9a shows an example of when a motion planned trajectory (*solid line*) exceeds the maximum curvature of a vehicle, $\kappa_{\max}$. Figure 9b shows an example of the vehicle (*dashed line*) trying to follow the planned trajectory (*dash-dot-dot line*) but is unable to match the curvature of

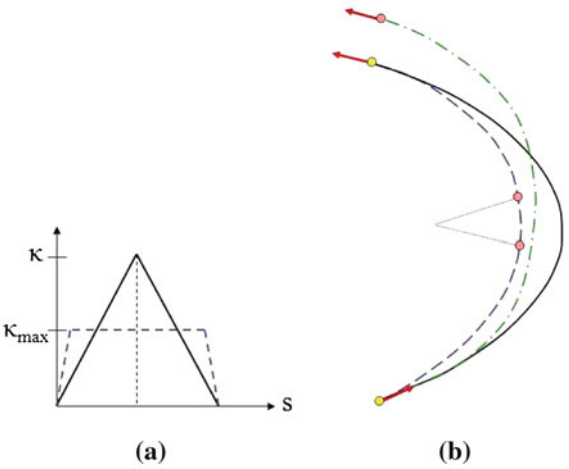


Fig. 9 The *solid lines* shows a motion planned trajectory with curvature exceeding κ_{max}. The *dash-dotted lines* show the path a vehicle would make while trying to follow the black trajectory. The *dashed lines* show a motion planned trajectory that the vehicle could follow with $\kappa = \kappa_{max}$. The paths are in the **a** $s\kappa$ domain and **b** in the xy domain

the trajectory because the curvature of the trajectory exceeds the maximum curvature the vehicle can achieve. The result is that the vehicle does not end at the desired ending waypoint.

However, by inserting an arc with $\kappa_{arc} = \kappa_{max}$ into the motion planned trajectory between the two clothoids, the vehicle is able to follow the path because the new trajectory does not exceed κ_{max}. Thus, the vehicle follows the path and ends at the desired ending waypoint.

To create a clothoid set with an arc is a process similar to that of creating a clothoid set without an arc. To develop a symmetric clothoid-arc set, first create a clothoid with a maximum curvature of κ_{max}. Then add an arc to finish the remaining half of the distance between the beginning and ending points. The arc is then mirrored and a mirror of the first clothoid is added on the end (Fig. 8).

Figure 7a shows a curvature versus distance plot for a two-clothoid trajectory with unrestrained curvature. If the maximum curvature of the two clothoids is greater than κ_{max}, the maximum curvature that the vehicle is able to drive, then the vehicle would be unable to follow the trajectory. Figure 8 shows that inserting an arc between the two clothoids with a radius, $r = \frac{1}{\kappa_{max}}$, results in a trajectory that the vehicle can follow.

If the curvature of the planned clothoid set exceeds κ_{max} then an arc is inserted between the beginning and ending clothoids. In order to insert an arc, the angle of the arc, λ, length of the arc, L_{arc}, and the endpoint of the first clothoid must be determined. In the derivation that follows we find a method of computing λ.

Fortunately, it is advantageous to start with the change of heading, length, and curvature of the clothoid set already planned. Because the desired change in heading has not changed, the area δ in Fig. 7a must be equal to the area of half the trajectory in Fig. 8a. In other words, the area for δ in Fig. 7a is

$$\delta = \frac{\kappa L_c}{2} \tag{16}$$

and the area for half of the trajectory, i.e., one clothoid and half the arc in Fig. 8a is

$$\delta = \frac{\kappa_{\max} L_{c_1}}{2} + \lambda \tag{17}$$

where λ, the angle of the circular arc, is

$$\lambda = \kappa_{\max} L_{\text{arc}}. \tag{18}$$

Next, set Eq. 16 equal to Eq. 17.

$$\frac{\kappa L_c}{2} = \frac{\kappa_{\max} L_{c_1}}{2} + \kappa_{\max} L_{\text{arc}}. \tag{19}$$

Also from Fig. 8a it can be seen that

$$L = L_{c_1} + L_{\text{arc}}. \tag{20}$$

Using Eq. 20 multiply both sides of the equation by $\kappa_{\max}$, and subtract the result from Eq. 19 and solve for L_{c_1}.

$$L_{c_1} = 2L - L_c \frac{\kappa}{\kappa_{\max}}. \tag{21}$$

Next, solve for L_{c_1} in Eq. 17 and set the result equal to Eq. 21.

$$\frac{2(\delta - \lambda)}{\kappa_{\max}} = 2L - L_c \frac{\kappa}{\kappa_{\max}}. \tag{22}$$

Finally, solve for λ in Eq. 22.

$$\lambda = 2\delta - \kappa_{\max} L. \tag{23}$$

In the results above, λ is half the angle of the total circular arc that is to be inserted into the trajectory. However, λ, L, L_1, and L_3 are all unknown and are dependent upon each other, thus, an approximation must be made to calculate λ. In the following section, a function $G(\lambda)$ is derived whose root is the correct λ. As Clothoid_Turn() returns κ, L_c, and α, let $L = L_c$ to find an approximation for λ. Then by using Newton's root approximation method, similar to Walton and Meek [13], the correct values for λ and L can be determined. Once λ and L have been found, a trajectory can be created for a clothoid with length L_{c_1}, an arc with length $2L_{\text{arc}}$ and angle 2λ, and finishing with a mirrored clothoid (Fig. 8).

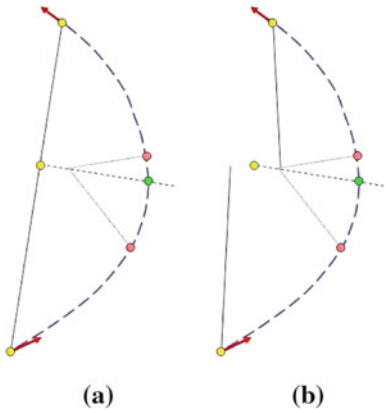


Fig. 10 **a** Shows that the ending point of the first clothoid-arc set and the beginning of the second set are not parallel to *d*. **b** Shows that by forcing line *d* of the first clothoid-arc set to be parallel with its endpoint, and the line *d* of the second set to be parallel with its beginning point, creates a discontinuous polyline

3.3 Asymmetric Clothoid-Arc Set

In previous papers we discussed the possibility of an asymmetric clothoid-arc set [6, 5]. However, further research has shown that this is not possible using our method of creating clothoid-arc sets as described above. Using our method, we require that the ending point of the first clothoid-arc set and the beginning point of the second clothoid-arc set be parallel to line *d*. Figure 10a shows an asymmetric clothoid-circular arc curve. In the figure the ending point of the first clothoid-arc set and the beginning of the second set are not parallel to *d*, thus, in violation of our parallel requirement.

Furthermore, if we were to force the line *d* of the first clothoid-arc set to be parallel with its ending point and the line *d* of the second clothoid-arc set to be parallel with its beginning point, the polyline then becomes discontinuous and therefore is no longer a polyline (Fig. 10b).

Using a combination of straight lines, clothoids, and arcs, trajectories are generated that are smooth, natural, and drivable. Arcs are used to provide an upper bound on curvature such that the maximum curvature of a vehicle is not exceeded. Clothoids are used to make the transitions between straight lines and arcs smooth and drivable. The next step is to create maneuvers from the trajectories to be used by the vehicle.

4 Application of Trajectories

This section discusses some trajectories created for common driving maneuvers of a DARPA Urban Challenge vehicle using clothoid-arc sets and straight lines. Each of the trajectories follow the specifications that a trajectory begins and ends with zero curvature, the vehicle's minimum turning radius (i.e., maximum

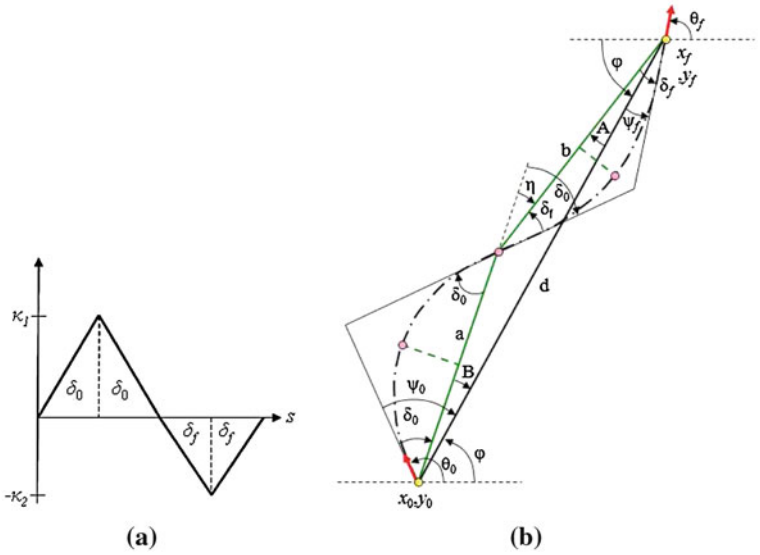


Fig. 11 Generic turn trajectory in the **a** $s\kappa$ plane and **b** xy plane. (Note that line d here is different from line d in Fig. 7b)

curvature, $\kappa_{\max}$) is not violated, and the only information given to the system are the initial and final points and their associated headings.

4.1 Generic Turn Trajectory

Clothoid sets or clothoid-arc sets can only provide left-turn, right-turn, and u-turn (u-turns are discussed in Sect. 2) trajectories. However, one of the purposes of the motion planner is to develop paths between any two points and headings provided by the path planner. Because the Clothoid_Turn() algorithm uses only one constructive polyline, it cannot generate maneuvers that require more than one curve between two points. The purpose of the generic turn is to use two constructive polylines to create a trajectory between any two given points with any two headings, assuming there is sufficient distance between the points that a trajectory can be created within the constraints stated in the introduction above.

Given the total change in heading and the length of line between the two waypoints, the algorithm developed below calculates the length of two constructive polylines and the change of headings associated with the clothoid sets for the two constructive polylines (Fig. 11). Then the generic turn algorithm calls the Clothoid_Turn() algorithm for each of the two calculated constructive polylines and the change of headings for the clothoid sets.

4.1.1 Generic Turn Algorithm

Figure 11 shows a generic turn in the $s\kappa$ plane and in the xy plane. To create a trajectory between the initial point x_0, y_0 and the final point x_f, y_f, the lengths of two constructive polylines a and b, and the deflection angles δ_0 and δ_f of the first and second clothoid-arc sets, respectively, must be computed (Fig. 11). Where polylines a, b, and line d are the sides of a triangle formed by the initial and final points and a midpoint (Fig. 11b). The only input variables of the system are initial and final points x_0, y_0, x_f, and y_f and their associated headings θ_0 and θ_f, respectively. From these inputs, values for d, ψ_0, ψ_f, η, and φ are found using Eqs. 27–29, 24, and 15, respectively. Where d is the length of line between the initial and final points, ψ_0 is the angle between the heading of the initial point and line d, ψ_f is the angle from the same line d and the heading of the final point, η is the direction angle between adjacent polylines a and b, and φ is the angle from the initial point to the final point (along line d). (It is important to note that half of the lengths of a and b passed to the Clothoid_Turn() algorithm as the constructive polylines to scale clothoids and generate the clothoid sets in Fig. 11b.) Using the geometry in Fig. 11b, equations for constructive polylines a and b are derived using the Law of Sines. Equations 25–30 are also obtained from the figure and are used to compute values for angles δ_0 and δ_f.

$$\varphi = \operatorname{atan}\left(\frac{y_f - y_0}{x_f - x_0}\right), \tag{24}$$

$$\theta_0 = \varphi - B - \delta_0, \tag{25}$$

$$\theta_f = \varphi + A + \delta_f, \tag{26}$$

$$\psi_0 = \varphi - \theta_0 = \delta_0 + B, \tag{27}$$

$$\psi_f = \theta_f - \varphi = \delta_f + A, \tag{28}$$

$$\eta = A + B = \delta_0 + \delta_f. \tag{29}$$

For the first clothoid set in the generic turn, the deflection is labeled as δ_0 and the second clothoid set deflection is δ_f. Thus, the total deflection from the initial point to the final point is the sum of all angles δ_0 and δ_f (30).

$$\theta_f - \theta_0 = 2\left(\delta_0 + \delta_f\right) = 2\eta. \tag{30}$$

Because A, B, δ_0, and δ_f cannot be computed from Fig. 11b and Eqs. 11, they must be fixed before the construction polylines a and b can be determined. Table 1 is used to summarize the constraints of Eqs. 25–29 that restrict the values for A, B, δ_0, and δ_f. The sums of rows and columns, η, and ψ_0, ψ_f, respectively, are functions of inputs in Eqs. 24, 27–29). The horizontal rows sum to $A + B = \eta$

Table 1 Constraints of Eqs. 25–29

A	B	η
δ_f	δ_0	η
ψ_f	ψ_0	2η

Table 2 Generic turn with null space parameter q added to constraints

$A+q$	$B-q$	η
δ_f-q	δ_0+q	η
ψ_f	ψ_0	2η

(29); $\delta_0 + \delta_f = \eta$ (29); and $\psi_0 + \psi_f = 2\eta$. The vertical columns sum to $A + \delta_f = \psi_f$ (28); $B + \delta_0 = \psi_0$ (27); and $\eta + \eta = 2\eta$.

While Table 1 constrains the variables A, B, δ_0, and δ_f, there remains one degree of freedom in the system, $A + B = \delta_0 + \delta_f$. To characterize an entire family of solutions, a null space parameter q is introduced. By adding a null space to Table 1 it allows A, B, δ_0, and δ_f, to be adjusted with respect to each other while maintaining the integrity of the constraints. Table 2 shows that given a solution (A, B, δ_0, δ_f) then ($A + q$, $B - q$, $\delta_0 + q$, $\delta_f - q$) is also a solution for any value of q.

To show that a null space of Table 1 does not change the solution of the constraints, let the constraints of Table 1 be put into matrix form (31).

$$M_c \begin{bmatrix} A \\ B \\ \delta_f \\ \delta_0 \end{bmatrix} = \begin{bmatrix} \eta \\ \eta \\ \psi_f \\ \psi_0 \end{bmatrix}, \quad \text{where } M_c = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}. \tag{31}$$

The system null space is found to be

$$M_c \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = 0. \tag{32}$$

Adding the null space, multiplied by parameter q, to the results of Eq. 31 (see Eq. 33) results in:

$$M_c \left(\begin{bmatrix} A \\ B \\ \delta_f \\ \delta_0 \end{bmatrix} + q \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \right) = M_c \begin{bmatrix} A \\ B \\ \delta_f \\ \delta_0 \end{bmatrix} + qM_c \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} \eta \\ \eta \\ \psi_f \\ \psi_0 \end{bmatrix}. \tag{33}$$

Thus, the free parameter q characterizes an entire family of solutions given valid values for A, B, δ_0, and δ_f. Although any number of solutions satisfy the equations in Table 1, Eq. 29 immediately suggests the candidate solutions: ($A = 0$, $B = 0$), ($A = 0$, $B = \delta_0 + \delta_f$), ($A = \delta_0 + \delta_f$, $B = 0$), ($A = \delta_0$, $B = \delta_f$), and ($A = \delta_f$, $B = \delta_0$). The solution $A = 0$ and $B = 0$ produces a straight line with a

Table 3 Constraints for the solution $A = \delta_0$ and $B = \delta_f$

δ_0	δ_f	$\delta_0 + \delta_f = \eta$
δ_f	δ_0	η
ψ_f	ψ_0	2η

Table 4 Constraints for the solution $A = \delta_f$, $B = \delta_0$, and free parameter q

$\frac{\psi_f}{2} + q$	$\frac{\psi_0}{2} - q$	$\frac{\psi_0 + \psi_f}{2} = \eta$
$\frac{\psi_f}{2} - q$	$\frac{\psi_0}{2} + q$	$\frac{\psi_0 + \psi_f}{2} = \eta$
ψ_f	ψ_0	$\psi_0 + \psi_f = 2\eta$

length of $a + b$. The second and third solutions are impossible because they would require that one of the angles of the triangle formed by lines *abd*, of which A and B are angles of, be zero. Both trajectories are members of the generic turn family and are discussed in Sect. 2. This leaves the two candidate solutions: $\left(A = \delta_0, B = \delta_f\right)$ and $\left(A = \delta_f, B = \delta_0\right)$. By substituting the first remaining candidate solution into Eqs. 25–29, the following equations are produced.

$$\theta_0 = \varphi - \delta_0 - \delta_f, \theta_f = \varphi + \delta_0 + \delta_f, \eta = \delta_0 + \delta_f, \tag{34}$$

$$\psi_0 = \delta_0 + \delta_f, \psi_f = \delta_0 + \delta_f. \tag{35}$$

Table 3 summarizes the constraints for the solution $A = \delta_0$ and $B = \delta_f$. Summing the columns and rows of the table show that the solution is valid, but only when $\psi_0 = \psi_f$ (35). As the matrix of Table 1 is rank deficient (31), one of the dependent rows can be removed and the new constraint, $\psi_0 = \psi_f$, added to form a matrix for Table 1.

The constraint $\psi_0 = \psi_f$ added to the other constraints of the table results in a full rank system and is therefore a single point solution.

The next step is to check whether the remaining candidate solution, $\left(A = \delta_f, B = \delta_0\right)$, is valid and provides a desirable family of solutions. This solution reduces Eqs. 25–29 to the following equations.

$$\theta_0 = \varphi - 2\delta_0, \tag{36}$$

$$\theta_f = \varphi + 2\delta_f, \tag{37}$$

$$\psi_0 = 2\delta_0, \tag{38}$$

$$\psi_f = 2\delta_f, \tag{39}$$

$$\eta = \delta_0 + \delta_f. \tag{40}$$

Again, Table 4 summarizes the constraints for the solution $\left(A = \delta_f, B = \delta_0\right)$, and free parameter q. Each column and row of the table sum correctly, therefore it

is a valid solution. Furthermore, there are no other additional constraints introduced to the system and the matrix of Table 4 is identical to the matrix of Table 1 which is rank deficient with a rank of 3. Thus, the solution $A = \delta_f$ and $B = \delta_0$ produces a family of valid solutions for any value of free variable q.

It should be noted that q needs to be constrained by A and B. As A and B are always the same sign due to the Law of Sines and assuming forward motion trajectories only; thus, q cannot be so large, either positive or negative, as to cause the angles $A + q$ and $B - q$ to have opposite signs. Therefore, when $A + q \geq 0$ and $B - q \geq 0$ then q is constrained by $B \geq q \geq -A$. And when $A + q \leq 0$ and $B - q \leq 0$, then q is constrained by $-A \geq q \geq B$.

Finally, after determining a valid solution for A and B and using Eqs. 38 and 39, δ_0 and δ_f can be computed, and therefore polylines a and b can be found for each solution within the solution set using the Law of Sines (Eqs. 41, 42) on triangle *abd* in Fig. 11b. With the length of constructive polylines a and b computed, the generic turn problem reduces to two clothoid set trajectories. Figure 12 outlines the algorithm for the generic turn.

$$a = \frac{d\sin|A|}{\sin|\pi - \eta|} = \frac{d\sin\left|\frac{\psi_f}{2} + q\right|}{\sin|\eta|} \tag{41}$$

$$\text{and} \quad b = \frac{d\sin|B|}{\sin|\pi - \eta|} = \frac{d\sin\left|\frac{\psi_0}{2} - q\right|}{\sin|\eta|}. \tag{42}$$

4.1.2 Finding q for Generic Turn

Now that the generic turn algorithm has been defined and q constrained, it must be determined what values of q should be passed to the algorithm. One solution is to let polylines $a = b$ and find the value of q that is valid for $a = b$. Once again, it should be noted that letting $a = b$ is but one solution and other ratios are also valid. However, letting $a = b$ ensures that if one polyline is long enough to create a clothoid-arc set the other polyline is too. Thus, by setting Eqs. 41 and 42 equal to each other, they reduce to

$$\sin\left|\frac{\psi_f}{2} + q\right| = \sin\left|\frac{\psi_0}{2} - q\right|. \tag{43}$$

To solve for q, the arcsine is taken on both sides,

$$a\ \sin\left(\sin\left|\frac{\psi_f}{2} + q\right| + n2\pi\right) = a\ \sin\left(\sin\left|\frac{\psi_0}{2} - q\right| + m2\pi\right) \tag{44}$$

where n and m are integers. Equation 44 then reduces to

Generic_Turn($x_0, y_0, \theta_0, x_f, y_f, \theta_f, q$)

{

$\varphi = \text{atan}\left(\frac{y_f - y_0}{x_f - x_0}\right)$

$d = \sqrt{(x_f - x_0)^2 + (y_f - y_0)^2}$

$\psi_0 = \varphi - \theta_0$

$\psi_f = \theta_f - \varphi$

$\eta = \psi_0 + \psi_f$

$A = \frac{\psi_f}{2} + q$ (see Table 4)

$B = \frac{\psi_0}{2} - q$ (see Table 4)

$\delta_0 = \frac{\psi_0}{2} + q$ (see Table 4)

$\delta_f = \frac{\psi_f}{2} - q$ (see Table 4)

$a = \frac{d \sin(|A|)}{\sin(|\eta|)}$

$b = \frac{d \sin(|B|)}{\sin(|\eta|)}$

Clothoid_Turn$\left(\frac{a}{2}, \delta_0\right) \rightarrow \kappa_1, \alpha_1, L_1$

Clothoid_Turn$\left(\frac{b}{2}, \delta_f\right) \rightarrow \kappa_2, \alpha_2, L_2$

Trajectory$\leftarrow$clothoid$(\kappa_0 = 0, \alpha = \alpha_1, L = L_1)+$
clothoid$(\kappa_0 = \kappa_1, \alpha = -\alpha_1, L = L_1)+$
clothoid$(\kappa_0 = 0, \alpha = \alpha_2, L = L_2)+$
clothoid$(\kappa_0 = \kappa_2, \alpha = -\alpha_2, L = L_2)$

}

Fig. 12 Algorithm for a symmetric generic turn trajectory. The output of the generic turn algorithm is a trajectory of four concatenated clothoid turns

$$\left|\frac{\psi_f}{2} + q\right| + n2\pi = \left|\frac{\psi_0}{2} - q\right| + m2\pi. \tag{45}$$

Due to the nature of absolute values there are two solutions to the equality, Eqs. 46 and 47.

$$-\frac{\psi_f}{2} - q + n2\pi = \frac{\psi_0}{2} - q + m2\pi \rightarrow -\frac{\psi_f}{2} = \frac{\psi_0}{2} + n'2\pi \tag{46}$$

$$\text{and}\,\frac{\psi_f}{2}+q+n2\pi=\frac{\psi_0}{2}-q+m2\pi \overset{\text{Solve for } q}{\longrightarrow} q=\frac{\psi_0-\psi_f}{4}+n'\pi, \qquad (47)$$

where $n'=m-n$. Interestingly, the solution in Eq. 46 does not contain the parameter q. This is the case of a lane change which is discussed in Sect. 2. The second solution, Eq. 47, provides an equation for q for when polylines a and b are equal. The value of q is calculated and passed to Generic_Turn().

4.2 Special Case Maneuvers

The DARPA urban challenge competition requires that an autonomous vehicle perform lane changes and u-turns [2]. This section shows how to create specific maneuvers, lane change, curve-straight, straight-curve, and u-turns for an urban challenge vehicle.

4.2.1 Trajectories for Turns and Straight Segments

Sometimes a clothoid set produces a trajectory that, although valid, is undesirable, such as in the case of making a shallow left-turn at an intersection. For example, if the vehicle on the left in Fig. 13a stops at the left part of the intersection, the motion planner should not plan a trajectory such that the autonomous vehicle at the bottom of the figure collides with the vehicle on the left. For these types of situations, it is easier to have special case trajectories that have a straight line either before or after a clothoid set. Figure 13b shows that the vehicle on the left can be avoided by adding a straight line after a clothoid set left-turn.

The generic turn method can produce a straight segment along polyline a, followed by one clothoid set using polyline b (Fig. 14a) by choosing a specific member of the solution set (i.e., by choosing q). This is called a straight-curve trajectory and it is produced by setting $q=\frac{-\psi_0}{2}$. Applying q to the generic turn, Table 4, results in a straight-curve trajectory, Table 5, where $\delta_0=0$.

The generic turn method can also produce a trajectory consisting of a clothoid set on polyline a, followed by a straight segment of length b (Fig. 14b) by choosing another specific member of the solution set (i.e., by choosing a different q). This is called a curve-straight trajectory, and is produced by setting $q=\frac{\psi_f}{2}$. And again applying q to the generic turn, Table 4, results in a curve-straight trajectory, Table 6, where $\delta_f=0$.

Because the solutions of the maneuvers are members of the solution set of the generic turn, the algorithms for the straight-curve and curve straight maneuvers consist of a single call to Generic_Turn(), passing the appropriate value of q.

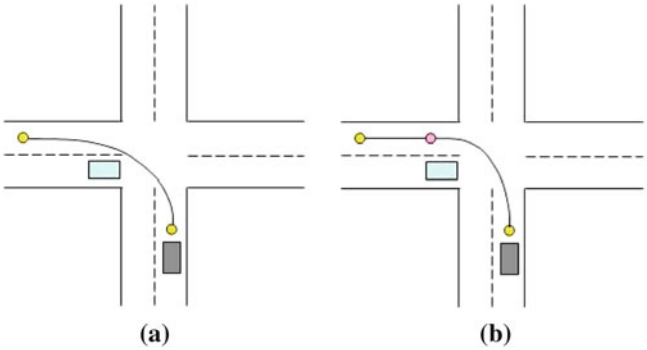


Fig. 13 **a** Undesirable shallow left-turn with a possible collision. **b** A left-turn consisting of a curve and a straight segment to avoid a collision

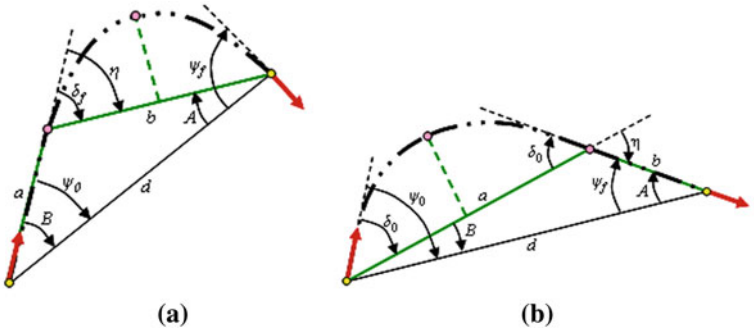


Fig. 14 **a** Straight-curve trajectory. **b** Curve-straight trajectory

Table 5 Constraints for the straight-curve trajectory for $q = \frac{-\psi_0}{2}$ (see Table 4)

$\frac{\psi_f}{2} - \frac{\psi_0}{2}$	ψ_0	$\frac{\psi_0+\psi_f}{2} = \eta$
$\frac{\psi_f}{2} + \frac{\psi_0}{2}$	0	$\frac{\psi_0+\psi_f}{2} = \eta$
ψ_f	ψ_0	$\psi_0 + \psi_f = 2\eta$

Table 6 Constraints for the curve-straight trajectory for $q = \frac{\psi_f}{2}$ (see Table 4)

ψ_f	$\frac{\psi_0}{2} - \frac{\psi_f}{2}$	$\frac{\psi_0+\psi_f}{2} = \eta$
0	$\frac{\psi_0}{2} + \frac{\psi_f}{2}$	$\frac{\psi_0+\psi_f}{2} = \eta$
ψ_f	ψ_0	$\psi_0 + \psi_f = 2\eta$

4.2.2 Lane Change Trajectory

One of the requirements of the DARPA urban challenge is that the vehicle must be able to pass another vehicle in the same lane if the second vehicle is stopped [2]; the lane change maneuver does just that. The trajectory is used when the beginning

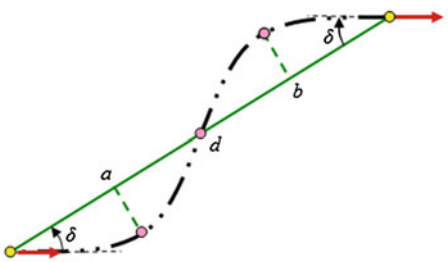


Fig. 15 Lane change trajectory maneuver

Table 7 Lane change trajectory constraints for no q and $\theta_0 = \theta_f$ (see Table 4)

0	0	$0 = \eta$
δ_f	δ_0	$0 = \eta$
ψ_f	ψ_0	$0 = 2\eta$

and ending point headings are the same (i.e., $\theta_0 = \theta_f$) but the points are offset by some x and y distance such that the vehicle can switch lanes on a roadway (Fig. 15.) This maneuver allows the path planner to provide two waypoints with the same headings. No change of heading means that $\eta = 0$. When $\eta = 0$, the Law of Sines cannot be used to calculate a and b due to division by zero.

Therefore, while q is passed to the Generic_Turn(), it is ignored for the case of the lane change and an exception must be made in the Generic_Turn() algorithm to handle the division by zero case. To avoid this it is assumed that $a = b = \frac{d}{2}$. While other distances for the polylines will also be valid, the distance of half of d helps ensure that lengths of polylines a and b are sufficiently large enough to create trajectories for them.

Because $\eta = 0$, the triangle formed by lines abd disappears. This then changes the angles that are passed to Generic_Turn(). As $A + q$ and $B - q$ are angles in that triangle, then $A + q = B - q = 0$. This means that q is ignored, and from Table 7 it can be seen that $\delta_0 = \psi_0$ and $\delta_f = \psi_f$. Figure 16 shows the changes made to the Generic_Turn() algorithm to handle the lane change exception.

4.2.3 *U*-Turn Trajectory

As stated previously, the u-turn is another maneuver required by the urban challenge. This trajectory comes about when the total deflection is approximately π. The actual trajectory is a mirrored clothoid set with d from Eq. 16 passed to a single call of Clothoid_Turn(). Only one call to Clothoid_Turn() implies that the u-turn has only one clothoid set, similar to a "C" path by Reeds and Shepp [11]. The trajectory consists of two clothoids (Fig. 17a) (i.e., clothoid ($\kappa_0 = 0, \alpha = \alpha, L = L$) + clothoid ($\kappa_0 = \kappa, \alpha = -\alpha, L = L$)) or two clothoids and an arc between them (Fig. 17 b) (i.e., clothoid ($\kappa_0 = 0, \alpha = \alpha, L = L_c$) + arc ($\kappa_0 = \kappa_{\max}, \alpha = 0$, $L = L_{\text{arc}}$) + clothoid ($\kappa_0 = \kappa_{\max}, \alpha = -\alpha, L = L_c$)).

Generic Turn($x_0, y_0, \theta_0, x_f, y_f, \theta_f, q$)

{

$\varphi = \text{atan}\left(\frac{y_f - y_0}{x_f - x_0}\right)$

$d = \sqrt{(x_f - x_0)^2 + (y_f - y_0)^2}$

$\psi_0 = \varphi - \theta_0$

$\psi_f = \theta_f - \varphi$

$\eta = \psi_0 + \psi_f$

if $\eta \neq 0$

{

$A = \frac{\psi_f}{2} + q$ (see Table 4)

$B = \frac{\psi_0}{2} - q$ (see Table 4)

$\delta_0 = \frac{\psi_0}{2} + q$ (see Table 4)

$\delta_f = \frac{\psi_f}{2} - q$ (see Table 4)

$a = \frac{d \sin(|A|)}{\sin(|\eta|)}$

$b = \frac{d \sin(|B|)}{\sin(|\eta|)}$

Clothoid Turn$\left(\frac{a}{2}, \delta_0\right) \rightarrow \kappa_1, \alpha_1, L_1$

Clothoid Turn$\left(\frac{b}{2}, \delta_f\right) \rightarrow \kappa_2, \alpha_2, L_2$

}

else //lane change

{

a=b=$\frac{d}{2}$

Clothoid Turn$\left(\frac{a}{2}, \psi_0\right) \rightarrow \kappa_1, \alpha_1, L_1$

Clothoid Turn$\left(\frac{b}{2}, \psi_f\right) \rightarrow \kappa_2, \alpha_2, L_2$

}

Trajectory$\leftarrow$clothoid($\kappa_0 = 0, \alpha = \alpha_1, L = L_1$)+

clothoid($\kappa_0 = \kappa_1, \alpha = -\alpha_1, L = L_1$)+

clothoid($\kappa_0 = 0, \alpha = \alpha_2, L = L_2$)+

clothoid($\kappa_0 = \kappa_2, \alpha = -\alpha_2, L = L_2$)

}

Fig. 16 Algorithm for a symmetric generic turn trajectory with added lane changes. The output of the algorithm is a trajectory of four concatenated clothoid turns

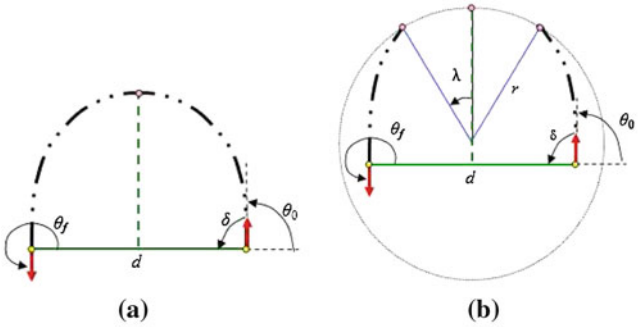


Fig. 17 **a** U-turn trajectory. **b** U-turn trajectory with an arc

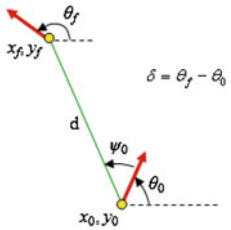


Fig. 18 Using ψ_0 and δ to determine which maneuver to execute

4.3 When to Execute a Trajectory Maneuver

The motion planner must know how to determine which maneuver to execute. One way is to examine the values and signs of ψ_0, ψ_f, and δ (as shown in Fig. 18. For example, for the maneuvers that have been discussed previously:

- If $\psi_0 > 0$ and $\delta > 0$ then a left-turn is performed,
- If $\psi_0 < 0$ and $\delta < 0$ then a right-turn is performed,
- If $\delta = 0$ then a left or right lane change maneuver is executed depending on the sign of ψ_0,
- If $\psi_0 = 0$ and $\text{sign}(\psi_0) = \text{sign}(\psi_f)$ then a curve-straight maneuver is performed; left or right turn depending on the sign of ψ_f,
- If $\psi_f = 0$ and $\text{sign}(\psi_0) = \text{sign}(\psi_f)$ then a straight-curve maneuver is performed; left or right turn depending on the sign of ψ_0,
- If $\psi_0 = \frac{\pi}{2}$ and $\delta = \pi$ then a left u-turn is executed.

It is important to note that the feasibility of a maneuver depends on the length of d. If d is too small, then it might not be possible to perform a given maneuver. (Imagine taking a *u*-turn to a point only two feet away.) Maneuvers depend upon d, ψ_0, and δ. It seems likely that the minimum length of d, for which a maneuver is feasible (46), is a function of ψ_0 and δ. This function is not given here, but it is a matter of future research.

$$d \geq f(\delta, \psi_0). \tag{48}$$

Maneuvers that cannot be performed are deemed "Illegal" meaning that either (1) a two-polyline trajectory for that maneuver cannot be created that is smooth and drivable, (2) the maneuver is not usually legal or wise to perform in normal traffic situations (e.g., a right hand *u*-turn), or (3) the maneuver is not simple and may require driving in reverse (e.g. a three-point turn). A further advantage of analyzing the legality of maneuvers is that it adds another check between the path planner and motion planner for drivability and safety. Should the motion planner encounter an illegal maneuver condition, it returns an error to the path planner stating that the motion planner cannot create a safe and drivable trajectory.

5 Results

The motion planner generates trajectories with the smallest curvature required to create a path between two waypoints. As stated earlier, trajectories generated by the motion planner are similar to paths that a human would drive and are unlike the Fraichard and Scheuer paths that have maximum sharpness and curvature [4]. As there is no standard metric to compare trajectories, one is proposed. The proposed metric is to compare the maximum curvature of a trajectory generated by both planners. The reason that this metric is reasonable is that we humans tend to turn the steering wheel of a vehicle as little as possible. For example, as the distance between two waypoints increases, humans turn the steering wheel less and less turning results in a smaller maximum curvature along a trajectory. Remember, that curvature is the inverse of the radius of the turn circle. Also, as the curvature decreases, the centrifugal force on the vehicle and on any on board passengers decreases (Eq. 49).

$$f \propto \kappa v^2, \tag{49}$$

where v is velocity of the vehicle or passenger and f is the centrifugal force. A smaller centrifugal force as a result of smaller curvature along a trajectory, not only allows for more passenger comfort (remember the teenager who always turns at maximum curvature) but also for higher velocities along a trajectory. Thus, the drivability and naturalness of the trajectories generated by both planners can be quantized by plotting the metric of maximum curvature along a trajectory with respect to the distance between the beginning and ending waypoints.

To obtain data for the metric, the maximum curvature of eight different left-turn trajectories is calculated for both the motion planner and the Fraichard and Scheuer method. The distance, d, between the origin and each of the end waypoints is shown in Table 8 and the waypoints for the trajectories are shown in Fig. 19. The left-turn trajectories all start at the origin and end at one of the other eight waypoints. The trajectories can all start at the origin without loss of

Table 8 The distance, d, between the origin and each of the end waypoints, x_f, y_f, in Fig. 19

End waypoint: x_f,y_f	Length of line d
−6, 6	8.49
−6, 10	11.66
−10, 6	11.66
−10, 10	14.14
−10, 15	18.03
−15, 15	21.21
−15, 20	25.0
−20, 20	28.28

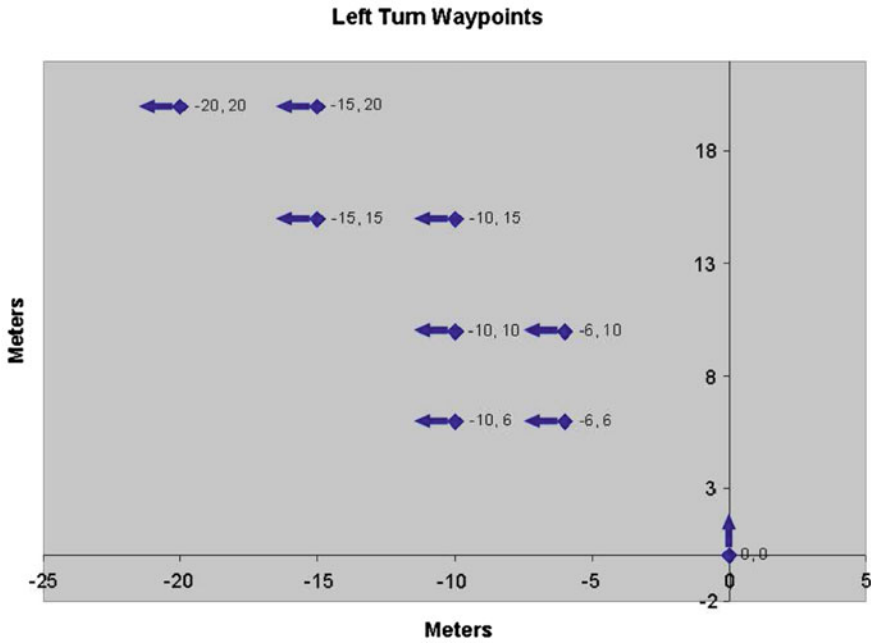


Fig. 19 Chosen waypoints for left-turn trajectories. Each left-turn starts at the origin and ends at one of the waypoints

generality. The maximum curvature of the DARPA urban challenge vehicle for which the motion planner was designed, is $0.2\frac{1}{m}$. Thus, the maximum curvature of both motion planning methods for this vehicle is $0.2\frac{1}{m}$ and is used as a basis for comparing the two methods.

The maximum curvatures of each left-turn trajectory versus the distance d of each left-turn trajectory for the motion planner and the Fraichard and Scheuer method are shown in Fig. 20. Remember, the Fraichard and Scheuer trajectories always have maximum curvature and sharpness, regardless of the type of trajectory. Thus, as the distance between the waypoints increases, the maximum

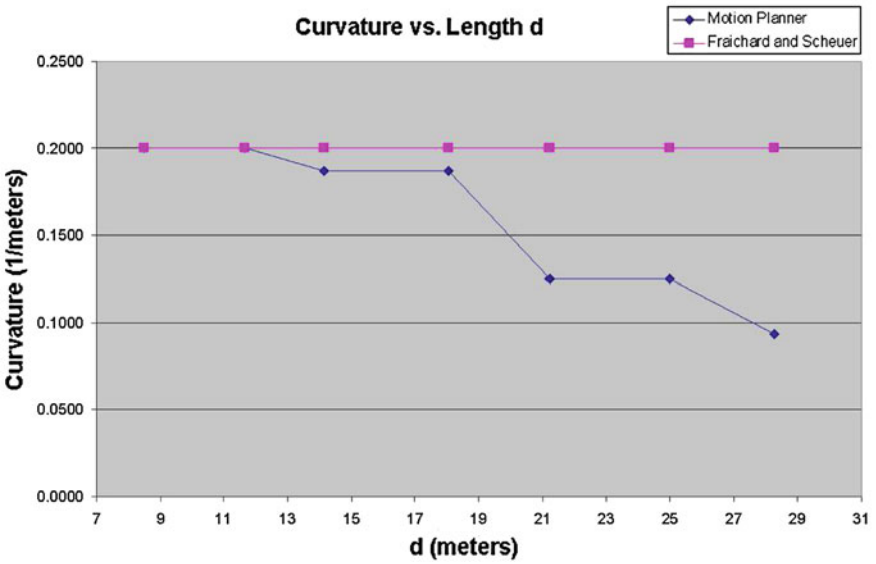


Fig. 20 Maximum curvature versus length of d for each of the left-turns in Fig. 19 for both the motion planner and Fraichard and Scheuer's method

curvature of each trajectory does not change. However, the motion planner presented in this chapter uses the smallest curvature required to generate a trajectory between two waypoints. Therefore, the maximum curvature of a trajectory decreases as the distance between the waypoints increases. The results in Fig. 20 show that the trajectories generated by the motion planner are more drivable and natural than those generated by the Fraichard and Scheuer method as the distance d increases.

5.1 Motion Planning a Test Course

Figure 21 shows a test course with hand picked waypoints to use each of the maneuver and trajectory algorithms discussed previously. The motion planner uses each of the maneuvers to plan paths between waypoints in the test course; straight, left-turn, right-turn, curve-straight, straight-curve, left and right lane changes, left u-turn, and the generic turn. The results of the motion planner after planning trajectories between each of the waypoints of the test course are shown in Fig. 21. Two more motion planner results are shown in Figs. 23 and 25 in Sect. 6.

This chapter has shown how maneuvers are formed from trajectories. An algorithm has been described that creates a trajectory that can begin at a point and heading and end at any point and heading given free variable q and sufficient distance between the points. Three other maneuvers (straight-curve, curve-straight,

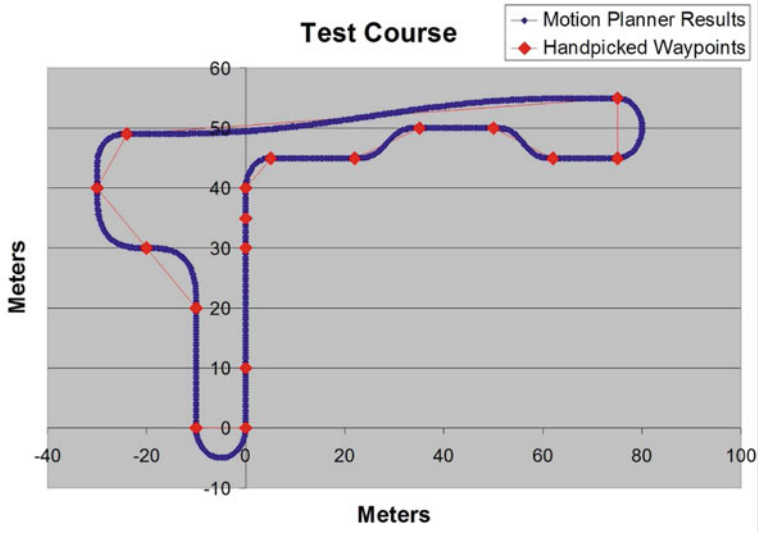


Fig. 21 Handpicked waypoints with the results of the motion planner after the waypoints are passed to the motion planner. The motion planner uses each of the maneuvers discussed previously to plan the example course; straight, left-turn, right-turn, curve-straight, straight-curve, left and right lane changes, left *u*-turn, and the generic turn. The trajectories of the motion planner are plotted every 0.25 m

and lane change) have been discussed and shown to be specific cases of the generic turn. It has also been explained when these maneuvers should be executed. A metric has been proposed and used to compare the motion planner trajectories and the Fraichard and Scheuer trajectories. The results show that the motion planner generates trajectories that are more drivable and natural than the Fraichard and Scheuer paths. The next chapter discusses a method of testing the motion planner and shows an example of a planned course.

6 Testing the Motion Planner

As with all software and systems, continuous testing of the motion planner and vehicle system is required throughout all stages of development. A set of waypoints can be used to provide a test course to test the abilities of the motion planner and the vehicle system. For example, Fig. 21 is a test course that tests each of the trajectories and maneuvers previously discussed. However, designing a test course by hand picking waypoints to pass to the motion planner is a laborious, time-consuming, and sometimes frustrating process. In addition to the difficulty of hand picking waypoints, testing the motion planner on the vehicle is more difficult than using a simulation at a desk. However, it is still important that the trajectories

produced by the motion planner be tested on the vehicle at sometime during the development of the vehicle. Henrie [5] describes an automated process called the Point Reduction Algorithm (PRA) that acts like a path planner and chooses a set of waypoints from a log file containing previously recorded GPS points and vehicle headings for each point. The following sections use the PRA to test the motion planner.

6.1 Example Tests of the Motion Planner

This section shows two examples of testing the motion planner using the PRA. In both examples, the log files contain real control data collected from a minivan using a GPS unit and corrected compass heading. (The compass heading needed to be corrected due to compass lag while turning.) The resulting trajectories from the motion planner in both examples are plotted every 0.25 m and are overlaid with the control data.

6.1.1 Test 1

Figure 22 shows an example plot of a log file containing over 2,400 waypoints recorded while manually driving the vehicle. The PRA chose only 29 points to describe the course, Fig. 22, and then passed these remaining waypoints to the motion planner. Figure 23 shows a plot of the results of the path planned by the motion planner with an overlay of the original log file. The results in Fig. 23 show that given the good input data from Fig. 22, the motion planner provides trajectories that deviate only slightly from the original path.

6.1.2 Test 2

Figure 24 shows a second example plot of a log file containing nearly 1,000 waypoints recorded while manually driving the vehicle. The PRA chose only 14 waypoints to pass to the motion planner, Fig. 24. Figure 25 shows a plot of the results of the path planned by the motion planner with an overlay of the original log file.

Unlike the results of the motion planner in the first test in Sect. 1, the results of the motion planner in Fig. 25 are somewhat poorer, in that the motion planned course significantly deviates from the test pattern. The deviation in the results from the original path could mean that either the motion planner planned a trajectory incorrectly or at least undesirably or the PRA poorly selected a waypoint. Further testing done by Henrie [5] shows that the PRA has selected poor waypoints to pass to the motion planner either due to a bug in the PRA algorithm or poorly recorded

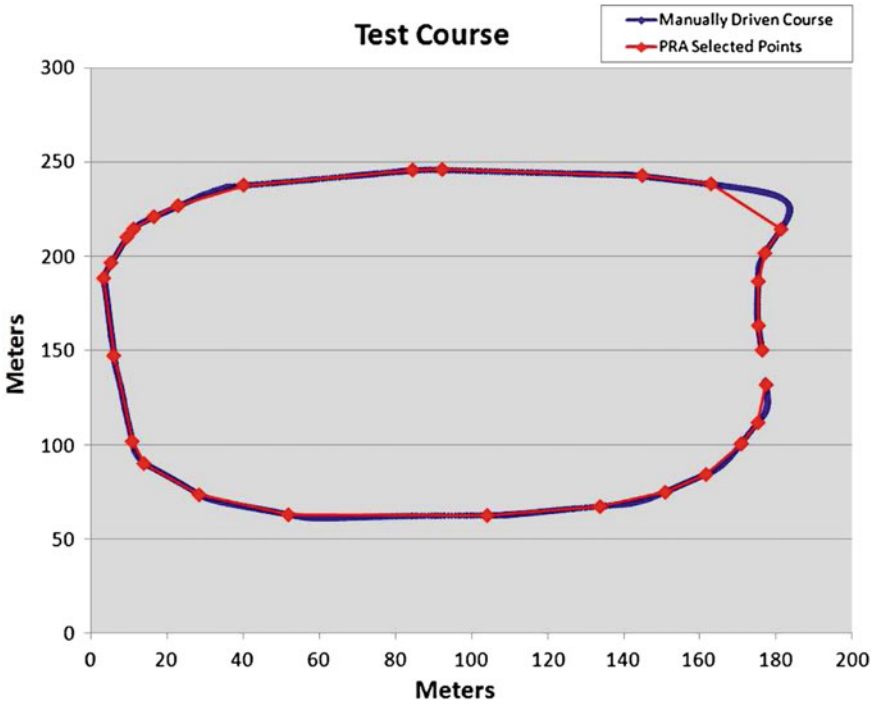


Fig. 22 A manually driven (thick line) course with over 2,400 waypoints, with the waypoints remaining after running the point reduction algorithm on the manually driven course waypoints. These remaining waypoints are passed to the motion planner to create a list of trajectories to recreate the course

waypoints. Poorly selected waypoints resulted in a less than desirable path generated by the motion planner (i.e., garbage in, garbage out).

In the end the motion planner did what it was supposed to do, plan a trajectory between two waypoints that is smooth, natural, and drivable. While some of the trajectories were not exactly desirable, it was a result of the PRA poorly choosing some waypoints and not the motion planner generating poor trajectories.

7 Conclusion

This chapter showed that the deflection δ of a clothoid determines its shape, and the curvature, κ determines its size. The x and y components of the clothoid curve, and hence the length of the curve, all scale linearly with $\frac{1}{|\kappa|}$ while the proportions of the clothoid curve and its deflection stay the same. Thus, given the deflection of a curve, a clothoid curve can be created, and then given the length of its associated

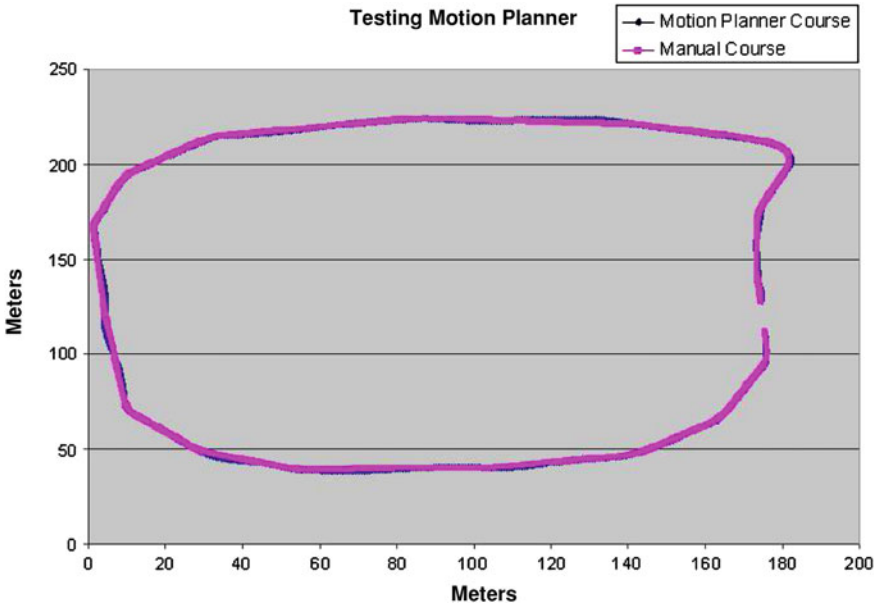


Fig. 23 Comparison of the first test manual driven course and the course created by the motion planner from the PRA selected points

polyline, the curve can be scaled to the correct size without changing its deflection. This principle has been used to design the Clothoid_Turn() primitive algorithm that allows us to generate clothoids using constructive polylines. Using this method, paths are computed between an initial point and final point using only the coordinates of those points and their associated headings. This is the basis of the motion planner.

Building upon the Clothoid_Turn() algorithm, circular arcs were included to limit the upper bound of the curvature of a clothoid turn. The curvature of the circular arc is the maximum curvature that the vehicle can physically turn. And the angle of the circular arc is found using Newton's root approximation and iterating until a root is found.

Using constructive polylines and the Clothoid_Turn() algorithm with circular arcs, a clothoid set has been developed consisting of two clothoid curves which is a turn similar to a "*C*" path by Reeds and Shepp [11], except that it uses clothoid arcs as well as circular arcs. The *u*-turn is a special case of this type of path.

A generic turn algorithm was also developed that produces a path consisting of two clothoid sets similar to a "CC" path by Reeds and Shepp [11]. It has also been shown that the straight-curve turn, the curve-straight turn, and the lane change are all special cases of the generic turn. All of the maneuvers discussed in Sect. 2 are smooth, natural, and drivable trajectories.

Finally, it was shown how a log file containing waypoints of a test course can be used to test the motion planner and vehicle system. The point reduction algorithm

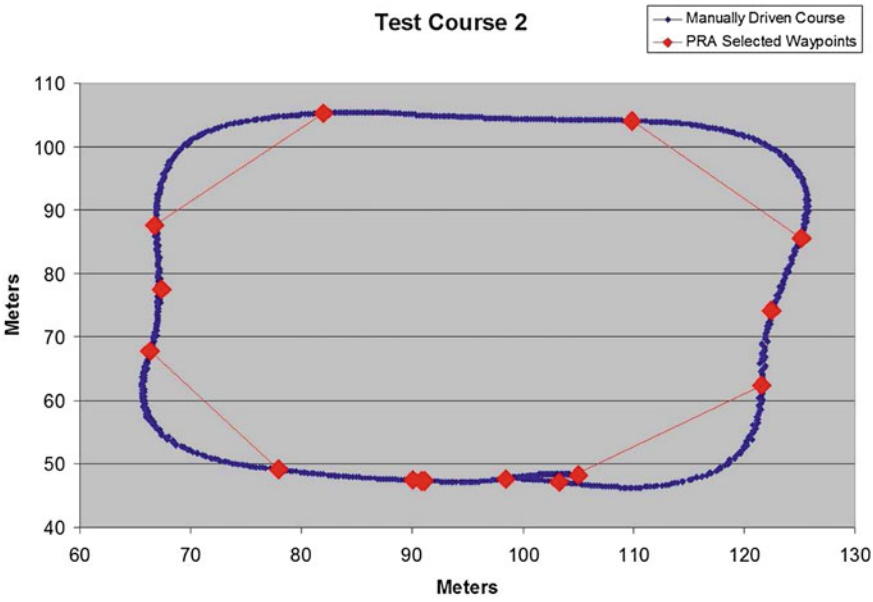


Fig. 24 A manually driven (thick line) course with nearly 1,000 waypoints, with the waypoints remaining after running the point reduction algorithm on the manually driven course waypoints. These remaining waypoints are passed to the motion planner to create a list of trajectories to recreate the course

selects a few waypoints from a log file and passes the selected waypoints to the motion planner. The motion planner is then tested to see if it can reproduce the test pattern. The vehicle system and controls can also be tested by driving the trajectories generated by the motion planner.

7.1 Future Research

Future research includes developing paths using the Clothoid_Turn() algorithm that include more than two polylines like those path sets described by Reeds and Shepp [11] and allowing for reversal in direction using free parameter q, and characterizing feasible paths by finding a function that will return the minimal length of d required for a valid trajectory (48).

The future work also includes the continued development of a set of heuristics to determine the type of maneuver to perform, how to parameterize the generic turn for different maneuvers, and to modify the point reduction algorithm to choose either previous or next waypoints from the list of zero and near-zero curvatures.

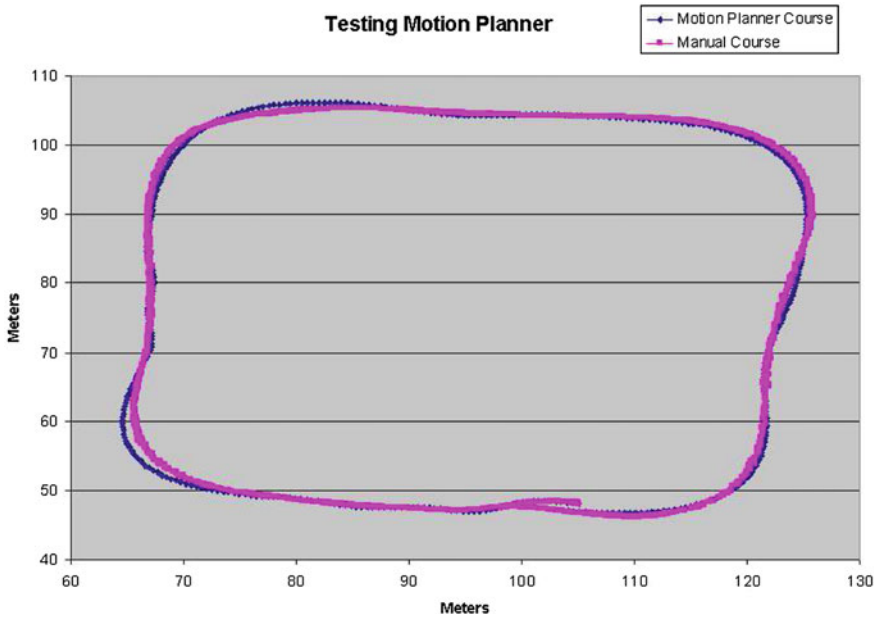


Fig. 25 Comparison of the manual driven course in Fig. 24 and the course created by the motion planner from the points in Fig. 24

References

1. Coombs, D., Murphy, K., Lacaze, A., Legowik, S.: Driving autonomously offroad up to 35 km/h. In: Proceedings of the IEEE Intelligence Vehicles Symposium, IEEE, pp. 186–191 (2000)
2. DARPA.: Urban Challenge Rules. Defense Advanced Research Projects Agency. http://www.darpa.mil/grandchallenge/rules.asp. Cited 31 October 2007
3. Dubins, L.: On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. Am. J. Math. **79**, 497–517 (1957)
4. Fraichard, T., Scheuer, A.: From Reeds and Shepp's to continuous-curvature paths. Robotics IEEE Trans. 20(6):1025–1035, doi:10.1109/TRO.2004.833789 (2004)
5. Henrie, J.: Planning continuous curvature paths using constructive polylines. Master's Thesis, Brigham Young University (2008)
6. Henrie, J., Wilde, D.: Planning continuous curvature paths using constructive polylines. J. Aerosp. Comput. Inf. Commun. **4**(12), 1143–1157 (2007)
7. Kimia, B.B., Frankel, I., Popescu, A.M.: Euler spiral for shape completion. Int. J. Comput. Vis. **54**(1–3), 157–180 (2003)
8. Knowles, D.: Real time continuous curvature path planner for an autonomous vehicle in an urban environment. Technical Report California Institute of Technology (2006)
9. Kolmanovsky, I., McClamroch, N.: Development in nonholonomic control problems. IEEE Control Syst. **15**(6), 20–36 (1995)
10. Pivtoraiko, M., Kelly, A.: A study of polynomial curvature clothoid paths for motion planning for car-like robots. Technical Report CMU-RI-TR-05-nn, The Robotics Inst., Carnegie Mellon University (2004)

11. Reeds, J., Shepp, L.: Optimal paths for a car that goes both forwards and backwards. Pac. J. Math. **145**(2), 367–393 (1990)
12. Shin, DH., Singh, S.: Path generation for robot vehicles using composite clothoid segments. Technical Report CMU-RI-TR-90-31, The Robotics Inst., Carnegie Mellon University (1990)
13. Walton, D., Meek, D.: A controlled clothoid spline. Comput. Graph. **29**(3), 353–363 (2005)

Part IV
Control and Sensors

Steering Controller of the CajunBot-II

Afef Fekih, Suresh Golconda, John Herpin and Arun Lakhotia

Abstract This chapter describes the steering controller of the CajunBot II, an autonomous vehicle designed for the DARPA Urban Challenge. The autonomous vehicle is a modified Jeep Wrangler Rubicon equipped with INS/GPS for localization, three sets of sensors and three single board computers. The steering controller is designed based on the look-ahead reference systems; a predictive control strategy where information about the heading angle and lateral displacement from the lane reference ahead of the vehicle is assumed to be available. The steering controller is designed to minimize error while following a desired path at a reasonable speed. Through the use of a lead compensator,we have been able to create a steering controller that tracks a variety of paths with reasonable performance and stability, even when these paths are such that the vehicle cannot track closely given its non-holonomic constraints. Additionally, through the use of a moving-average filer, we were able to increase the smoothness of the steering. The steering controller was tested under full-scale conditions and its dynamic performances are discussed here.

A. Fekih (✉) · J. Herpin
Electrical and Computer Engineering Department,
University of Louisiana at Lafayette,
P.O. Box 43890, Lafayette, LA 70504, USA
e-mail: afef.fekih@louisiana.edu

J. Herpin
e-mail: john.herpin@gmail.com

S. Golconda · A. Lakhotia
Center for Advanced Computer Studies,
University of Louisiana, lAFAYETTE,
P.O. Box 44330, Lafayette, LA 70504, USA
e-mail: suresh_golconda@yahoo.com

A. Lakhotia
e-mail: arun@louisiana.edu

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_8, © Springer-Verlag London Limited 2012

1 Introduction

The prime task of vehicle steering or lateral motion control is path/lane following, or more plainly, to keep the vehicle on the road/lane [2]. Vehicle lateral dynamics have been studied since the 1950s [5, 12, 15, 22] and are relatively easy to control; however designing and implementing a good steering controller requires a careful handling of several problems [16, 17, 18]. In general, vehicle lateral control strategies can be grouped into two kinds based on the employed sensors: look-ahead and look-down reference systems. Look-ahead reference systems resemble human driving by measuring the heading angle and lateral displacement from the lane reference ahead of the vehicle using vision sensors [11]. Look-down reference systems calculate the heading angle and lateral displacement from a reference point, for example a magnetic marker, very close to the vehicle [8].

Look-down reference systems have the advantage of being reliable, yielding accuracy and good performance under any weather or light conditions. And they can work well when other vehicles or pedestrians occlude lane lines. Therefore, it is very useful in urban areas navigation. However, it is also reported that look-down reference systems yield good experimental results only with speeds of less than 25 m/s (55 miles/h) which are impractical for highway automation. Look-ahead reference systems are suitable for such kind of driving scenarios, since current vehicle vision systems can effectively detect the upcoming road curvature variations and lateral displacement when driving at highway level 40 m/s (90 miles/h).

In this chapter, we discuss the steering controller of the CajunBot-II; an autonomous vehicle developed for the DARPA urban challenge [4]. It is a 2004 Jeep Wrangler Rubicon equipped with an INS/GPS for localization and a variety of environment sensors, stereo vision, and radars [14]. The CajunBot-II steering controller is designed based on the look-ahead reference systems. The control objective is to make the lateral error at a certain point ahead of the vehicle zero while following a reasonable path at a reasonable speed [3, 19]. The distance of this point from the vehicle is called the look-ahead distance. A proportional integral controller augmented with a moving-average filter to damp the yaw internal dynamics is proposed to achieve this objective. Test results under full-scale conditions show that the steering controller is able to track a variety of paths with reasonable performance and stability, even when these paths are such that the vehicle cannot track the desired path closely given its non-holonomic constraints. Further, the use of a moving-average filter had led to increased smoothness of the steering [9].

This chapter is organized as follows. Section 2 gives an overview of the CajunBot-II. The dynamic model of the vehicle is described in Sect. 3. The steering controller is discussed in Sect. 4. The performance of the controlled vehicle is presented in Sect. 5. Finally, Sect. 6 gives some concluding remarks.

Fig. 1 CajunBot-II

2 Overview of the CajunBot-II

CajunBot-II (Fig. 1) is an autonomous vehicle designed for the DARPA urban challenge. It is a 2004 Jeep Wrangler Rubicon equipped with an electronic drive-by-wire system from electronic mobility control (EMC). The jeep is fitted with 2 high-output alternators, each with maximum current output of 250 A to power all the computers, sensors, and electronics onboard. It gets its localization information from the RT3102 INS from oxford technology solutions. The INS is fed with a Starfire differential correction signals in RTCM format output by C&C C-Nav receiver for more accurate localization data. To put a bound to the GPS errors, the INS also receives input from a TTL level optical wheel encoder.

The CajunBot-II is equipped with a variety of environment sensors, to detect dynamic and static obstacles, including two SICK LMS LIDAR sensors, four Alasca XT LIDAR sensors from Ibeo, one set of stereo vision cameras, and four Eaton Vorad radars. These sensors provide data to the computing system through an ethernet interface or through a RS-232 connection.

CajunBot-II is equipped with three computers referred to as Main, NTP, and Logger. Each is a single board computer in EPIC form factor with a 1.8 GHz Pentium M. The Main computer is used for running autonomous software system, which includes modules such as drivers, path planner, and steering controller. The NTP computer provides the network time protocol service for the other computers. The Logger saves all the data acquired during a run and also has the ability to broadcast the data over a wireless network. The Main and Logger computers run Fedora Core 5 (FC5). The NTP service, although light-weight, is provided by a separate machine because the PPS patch needed for NTP is available for the Linux 2.4 kernel, and not for the Linux 2.6 kernel used in FC5. The Ibeo LIDAR system and Iteris LDW have their own computing units. All the computing devices, sensors, and other equipment communicate over ethernet, connected through a Gigabit Managed Switch from Dell.

Commands from CajunBot's computing system are actuated through a drive-by-wire system which was designed and installed by EMC. Through the drive-by -wire system, all of the functions normally controlled through the jeep's dashboard

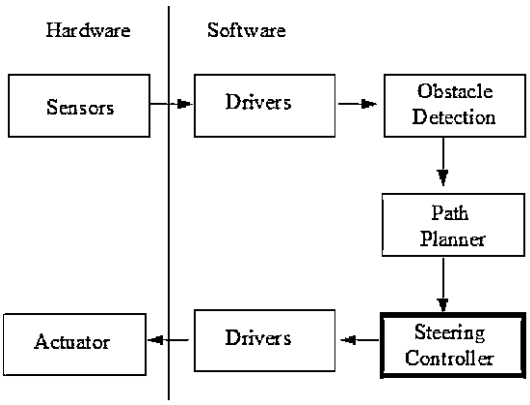


Fig. 2 Dataflow diagram of the CajunBot-II's autonomous software system

interface can be controlled. This includes, but is not limited to, starting the engine, turn signals, windshield wipers, and headlights. Additionally, the system actuates steering, throttle, brake, and gear changes.

The autonomous software system of CajunBot-II consists of five modules, namely Drivers, Obstacle Detection, Path Planner, Steering Controller, and Middleware. The Drivers provide the software-interface to the hardware. They read the raw sensor data and decoding it to the format that the rest of the software system can understand. It also communicates the software decision commands such as steering, throttle, and blinkers commands to the hardware-interface. The Obstacle Detection module is a collection of algorithms, each designed to analyze data and detect obstacles from one sensor type. The Path Planner [19] module takes as input the localization and the obstacles information and generates a path for the vehicle to follow, in order to achieve the given mission. The path generated here is a sequence of equally spaced GPS points that are annotated with maximum speed to drive through. The Steering Controller module generates the steering and throttle commands required to closely drive the CajunBot-II through the Path Planner's path while following a safe speed. And finally the Middle module provides a means for all the above software modules to communicate with each other. It is implemented as shared memory queues, with one writer and possible more than one reader. Figure 2 shows the data flow diagram of the autonomous software system.

The CajunBot software system includes three augmented software modules that facilitate development and testing of the autonomous software system. The augmented software modules include: Simulator, Visualizer, and Playback modules. The simulator module, built over a physic-based ODE engine provides the software simulation of the CajunBot-II and other elements from the real-world, such as the static obstacles and the traffic vehicles. Most of the autonomous software system is first developed and tested in the Simulator before even tried on the real vehicle. The visualizer module provides a graphical display of the outputs from all the software modules for visual analysis. The third augmented software module,

the playback allows retrieving data from the log files that are collected during an autonomous run. This retrieved data is used for post-run analysis and for debugging the updated software modules by running them over the retrieved data.

2.1 Steering Actuation System

The steering commands from the controller are actuated via a drive-by-wire system which was designed and installed by EMC. The steering and throttle are each actuated by a servo motor. The position of the motor is determined by an analogue voltage input. The motor will always move at a maximum constant speed until the position is reached or a new command is implemented. The steering and throttle commands, as implemented by the software, are values ranging from −1.0 to +1.0, which correspond to the minimum and maximum voltages, respectively. In the case of throttle, −1.0 is equivalent to a full brake while +1.0 is full throttle. In the case of steering, −1.0 is equivalent to a full left-turn while +1.0 is a full right-turn. The average ground wheel angle corresponding to a full left or right-turn is 28° off center. The ground wheel is assumed to change linearly with the steering wheel.

The limits of the actuation system play a significant role in the performance of the steering controller. The steering actuator introduces a delay of 0.24 s between the time when the steering command is implemented and when the actuator begins to move. Additionally, the steering actuator can only change the ground wheel angle at a rate of 15.6° s^{-1}. This corresponds to 3.6 s to go from an extreme left to extreme right-turn.

2.2 Navigator

The Navigator module takes as input the trail computed by the executive module. It is responsible for driving the vehicle along the sequence of way points in the trail, keeping the vehicle as close to the given path as possible, while also maintaining the speed specified in the trail. The Navigator uses a lead-lag compensator controller for controlling the vehicle's steering [10] and a PD controller for controlling its throttle and brakes.

The dynamic equations and the parameters of the vehicle are described in the following section.

3 Vehicle Dynamic Model

When the objective is to develop a steering control system for automatic lane keeping, it is useful to utilize a dynamic model in which the state variables are in terms of position and orientation error with respect to the road [1, 13].

The dynamic model of the vehicle in terms of the error with respect to the road is described by the following state space model [6].

$$\begin{cases} \dot{X} = AX + Bu \\ y = CX + Du \end{cases} \tag{1}$$

where,

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{\Omega_1+\Omega_2}{v} & \Omega_1+\Omega_2 & \frac{\Omega_1(d_s-l_f)+\Omega_2(d_s+l_r)}{v} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2(C_f l_f - C_r l_f)}{Jv} & \frac{2(C_f l_f - C_r l_r)}{J} & -\frac{2(C_f(l_f^2 - l_f d_s)+C_r(l_r^2 - l_r d_s))}{Jv} \end{bmatrix} \tag{2}$$

$$B = \begin{bmatrix} 0 & 0 \\ \Omega_1 & \frac{\Omega_2 l_r - \Omega_1 l_f - v^2}{v} \\ 0 & 0 \\ \frac{2C_f l_f}{J} & -\frac{2(C_f l_f^2 + C_r l_r^2)}{Jv} \end{bmatrix}, \tag{3}$$

$$C = [1 \quad 0 \quad d_s \quad 0], \quad D = [0 \quad 0] \tag{4}$$

$$\Omega_1 = 2C_f\left(\frac{1}{M} + \frac{l_f d_s}{J}\right), \quad \Omega_2 = 2C_r\left(\frac{1}{M} + \frac{l_r d_s}{J}\right) \tag{5}$$

with, $X = [y \quad \dot{y} \quad x \quad \dot{x}]^T$, the state vector, and $u = [\delta]^T$, the control vector.

J is the vehicle moment of inertia (kg*m^2); M is the vehicles mass (kg); l_f is the distance from the center of gravity to the front wheel axel (m); l_r is the distance from the center of gravity to the back wheel axel (m), and $L = l_f + l_r$; d_s is the distance from the center of gravity to the virtual sensor point (m); C_r is the rear cornering stiffness (N/rad); C_f is the front cornering stiffness (N/rad); v is the vehicle velocity (m/s); δ the steering angle; I the instantaneous yaw rate of the vehicle; y the lateral displacement of the virtual point; μ the road adhesion factor (from 1 to 0).

These parameters are illustrated in Fig. 3 and are readily available from the combination of GPS and INS for the CajunBot II. These parameters influence the steering controller which projects a *steering-target point, s*, in front of the vehicle and computes the *steering position*, δ, that would minimize the *lateral displacement, y*, of the steering-target point from the desired path.

4 Steering Controller

The steering controller is designed to minimize error while following a reasonable path at a reasonable speed [7]. Assuming that all paths and speeds fall well within the range of the vehicle's operating abilities; the steering controller is designed

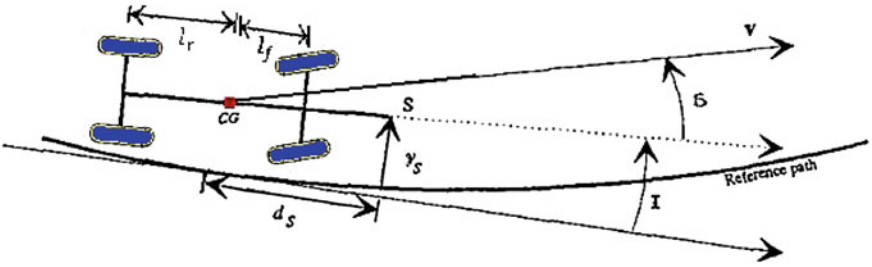


Fig. 3 Lateral vehicle dynamics

based on the look-ahead approach. Unlike look-down-controllers, which use the lateral deviation of one or more control points on the vehicle from the path, this method uses the lateral deviation from the path of a "virtual sensor" point at a specified distance in front of the vehicle as input and seeks to minimize this. The distance from the vehicle's center of gravity to this point is called the "look-ahead distance".

The steering controller projects a steering-target point in front of the vehicle and computes the steering position that would minimize the lateral displacement of the steering-target point from the desired path. The distance from the vehicle's center of gravity to the steering-target point is called the *steering-target or distance*. The transfer function from steering angle to lateral acceleration of the steering-target point is given by [20, 21]:

$$\frac{\ddot{y}(s)}{\delta(s)} = \frac{\mu C_f v^2 \left(M l_f d_s + I\right) s^2 + \mu^2 C_f C_r L v (d_s + l_r) s + \mu^2 C_f C_r L v^2}{(IMv^2) s^2 + \mu v \left(I\left(C_r + C_f\right) + M\left(C_f l_f^2 + C_r l_r^2\right)\right) s + \mu M v^2 \left(C_r l_r - C_f l_f\right) + \mu C_f C_r L^2} \tag{6}$$

By integrating Eq. 2, twice, we get the transfer function from ground wheel angle to lateral displacement of the virtual sensor point. Using this transfer function and unity feedback (assuming the sensor makes little or no changes to the measurement) the control law can be derived.

Using the root locus approach, a lead-lag compensator can be designed. The lead-lag compensator has the following standard form:

$$K(s) = K\left(\frac{s-a}{s-b}\right) \tag{7}$$

K is the gain of the system, $|b| > |a|$ for a lead compensator and $|a| > |b|$ for a lag compensator. The use of a lead compensator shifts the root locus of the system to the left, and thus increases the stability region and also increases the response time. The lag compensator is primarily used to improve the steady state performance of

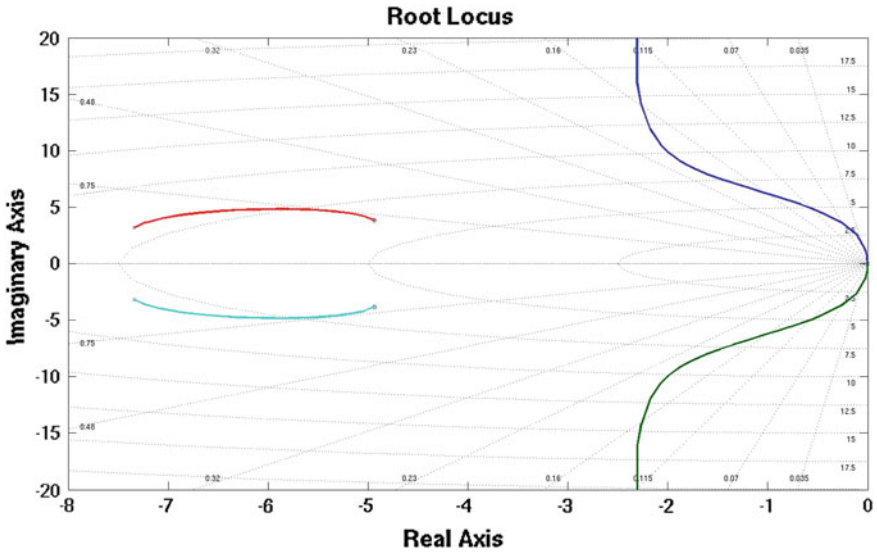


Fig. 4 Root locus plot

the system, that is, it helps reduce steady-state error. Because the primary concern for steering is stability and good tracking performance, we will focus mainly on the lead compensator.

The root locus of the function is shown in Fig. 4. The values of the design parameters a and b are computed in order to achieve a desired dampening ratio and natural frequency. This process is most commonly done by trial and error. Afterward, the gain is adjusted until maximum performance is observed. For CajunBot-II, the following parameters were used: $K = 3$, $a = -1$, and $b = -20$.

The steering position is described in the time domain by:

$$\delta(t) = P\varepsilon(t) * e^{(-20t)} + D\frac{d\varepsilon(t)}{dt} * e^{(-20t)} \tag{8}$$

where $*$ is the convolution of the two terms and $\varepsilon(t)$ is the error term. P and D are the proportional and derivative gains, respectively. The values for P and D are left as variables and used during implementation for tuning the controller.

5 Implementation

To validate the performance of the proposed look-ahead controller, the following is a series of experiments conducted for the CajunBot-II, under several conditions. The known parameters of the vehicle are illustrated in Table 1.

Table 1 Nominal parameters of the CajunBot II

Description	Parameter	Value	Units
Moment of inertia	J	8,470	kg m^2
Vehicle mass	M	2,000	kg
Vehicle velocity	v	15	m/s
Vehicle width	b	1.694	m
Distance from center of gravity to front axle	l_f	1.5	m
Distance from center of gravity to rear axle	l_r	1	m

In the implementation, the gains are being automatically tuned to compensate for model mismatch, system delays, and other characteristics caused by the digital implementation of the controller. Moreover, the steering-target distance is computed dynamically as a function of the vehicle's speed. What may be counter-intuitive, this distance decreases at higher speeds. A speed plan generates slower speed when the path has a greater lateral acceleration. The longer steering-target distance compensates for the steering delay of 0.24 s, thus preparing for the vehicle to turn ahead of time. On the other hand, at higher speeds a small change in the steering angle leads to larger change in the lateral displacement (for the same time interval), that is, the steering is more sensitive at higher speeds. A smaller steering-target distance at higher speeds ensures that the system generates gentle changes and provides greater stability. Presently, the look-ahead distance has a maximum value of 3.0 m from the front bumper of the vehicle. When the vehicle reaches a speed of 3.0 m/s the look-ahead distance begins to decline at a rate of 0.75 m per 1.0 m/s increase in speed until it reaches a minimum of 0.25 m from the front bumper.

Finally, the lateral displacement of the steering-target point y_s is computed as the distance from the steering-target point to the path, measured perpendicular to the vehicle's heading. When the vehicle is oriented near perpendicular to the path, such as at an intersection, the lateral displacement computed would be infinite, leading to an unpredictable behavior.

While the executive module generates smooth paths through the intersection, for reliability and to achieve algorithm independence, it is important that the steering controller achieves its expected tracking performance if the executive makes an error. One solution to this problem is to use a line perpendicular to the path segment to calculate the virtual sensor's lateral displacement. However, this too becomes troublesome as this can have multiple solutions and its computation is more complex. We use the following method for determining the lateral displacement. The steering-target point is determined by the normal method; however a second point, *vehicle track point*, is determined. This point is set by tracing the length of the steering-target distance along the vehicle's projected path. The distance between the vehicle-target point and the vehicle-track point is taken as the lateral displacement. For small heading errors, the difference in vehicle's heading and the path orientation, this computation yields values close to those from the previous method. However, when the heading error is large the new

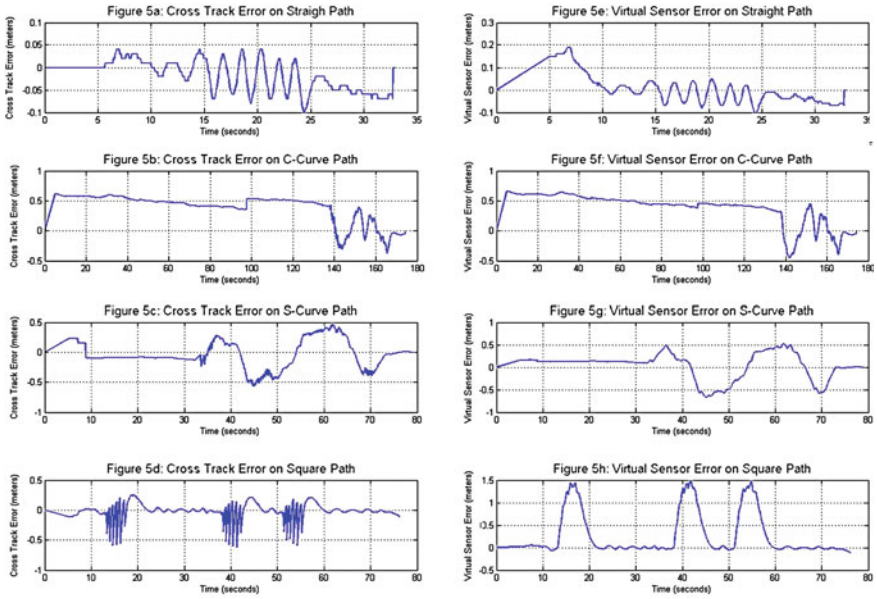


Figure 5a: Cross Track Error on Straigh Path
Cross Track Error (meters)
Time (seconds)
Figure 5e: Virtual Sensor Error on Straight Path
Virtual Sensor Error (meters)
Time (seconds)
Figure 5b: Cross Track Error on C-Curve Path
Cross Track Error (meters)
Time (seconds)
Figure 5f: Virtual Sensor Error on C-Curve Path
Virtual Sensor Error (meters)
Time (seconds)
Figure 5c: Cross Track Error on S-Curve Path
Cross Track Error (meters)
Time (seconds)
Figure 5g: Virtual Sensor Error on S-Curve Path
Virtual Sensor Error (meters)
Time (seconds)
Figure 5d: Cross Track Error on Square Path
Cross Track Error (meters)
Time (seconds)
Figure 5h: Virtual Sensor Error on Square Path
Virtual Sensor Error (meters)
Time (seconds)

Fig. 5 Cross-Track Error and Virtual Sensor Error for various paths

computation yields a stable solution. This method is also immune to variations in the path shape.

In order to account for small-scale oscillations (or jitters) in the steering and to minimize the negative effects of GPS shifts, a moving-average filter is added to the controller [23]. The filter is applied to both the input, virtual sensor error, and the output, steering angle. The filter is implemented by averaging any three points around a measured value and substituting the averaged value for the actual center value. Analytically, this can be viewed as forcing the values to change in a smooth manner while introducing a small delay of 0.05 s, or one interpolation at 20 Hz. This delay may seem counterproductive; however this delay is small when compared to the 0.24 s delay of the steering actuator, which is already accounted for in the look-ahead distance.

Figure 5 illustrates the cross track error as the vehicle drives over a variety of paths. The cross track error is calculated as the distance from the INS to the path measured perpendicular to the vehicles heading. Typically, cross track error is calculated as the distance to the path from the center of gravity of the vehicle; however the navigation data on the CajunBot-II is recorded as the location of the INS which is reasonably close to the center of the wheel base. For our purpose it serves as a reasonably close estimate.

While testing the steering performance, the vehicle was set to travel at a maximum speed of 12 m/s. Each curved path, Fig. 5b, c, was made by driving the intended path manually and collecting the navigation data. The path is then

extracted from this data in the form of 2 m long segments that approximate the curve. The square path was made of 4 straight segments connected by arcs or a circle.

Figure 5a shows the cross track error as the vehicle drives down a long straight path. The maximum cross track error is approximately 10 cm. It is worth noting that the GPS of CajunBot-II is only accurate up to 10 cm. Figure 5b shows the cross track error of the vehicle as it travels a long path curved in one direction. The vehicle reaches a maximum speed of 9 m/s. The track used for Fig. 5c was a S-shaped curve. On this path, the vehicle reaches a maximum speed of 7 m/s. On the square path from Fig. 5d, the vehicle reaches a maximum speed of 12 m/s on the straight segments and takes each turn at a speed of 3 m/s.

Figure 5 shows that the vehicle reaches a maximum cross track error of 0.6 m while maintaining stability. This error may be considered high by many standards, however, it is necessary in order to maintain stability and smoothness of steering on the wide variety of paths that CajunBot-II might encounter. For the tests shown in Fig. 5, the paths were made of either straight segments or made from splines collected from actually driving the vehicle. In the path planning process, many times, exact lane data in the form of splines is largely unavailable. In this case, the path is made of straight lines connecting way points. In this case, the angles between connecting segments can grow increasingly large, making it impossible for the vehicle to follow these paths with a reasonable cross track error while maintaining stability. In order to maintain stability in all situations and to further increase the smoothness of steering and control, the steering is tuned to not follow the path as closely as would be otherwise desirable.

6 Conclusion

A look-ahead steering controller was designed and implemented for an autonomous vehicle, with a primary aspect being to make the lateral error at a certain point ahead of the vehicle zero. Through the use of a lead compensator, we have been able to create a steering controller that tracks a variety of paths with reasonable performance and stability, even when these paths are such that the vehicle cannot track closely given its non-holonomic constraints. Additionally, through the use of a moving-average filer, we were able to increase the smoothness of the steering.

For the current implementation of path generation, the steering controller exhibits optimal performance. In the future, if more complete lane data becomes available, the controller may be tuned to track the path more closely. However, this tuning may lead to instability if the path has any discontinuous features.

Acknowledgments This work is partially supported by the University of Louisiana at Lafayette through the Louisiana Governor's Information Technology Initiative.

References

1. Beji, L., Bestaoui, Y.: Motion generation and adaptive control method of automated guided vehicles in road following. IEEE. Trans. Intell. Transp. Syst. **6**(1), 113–123 (2005)
2. Bishop, R.: Intelligent vehicle applications worldwide. IEEE. Intell. Syst. Appl. **15**(1), 78–81 (2000)
3. Chen, C., Tanm, H.S.: Steering control of high speed vehicle: dynamic look ahead and yaw rate feedback. In: Proceeding of the 37th IEEE Conference on Decision and Control. pp. 1025–1030, Tampa, Florida (1998)
4. DARPA Grand Challenge Race [online]. http://www.darpa.mil/grandchallenge
5. Fenton, R.E, Melocik, G.C, Olson, K.W: On the steering of automated vehicles: theory and experiment. IEEE. Trans. Autom. Control **21**, 306–315 (1976)
6. Gillespies, T.D.: Fundamentals of vehicle dynamics. Society Autom Eng, Warrendale (1992)
7. Guldner, J., Tan, H.S, Patwardhan, A.: Study of design directions for lateral vehicle control. In: Proceedings of the IEEE Conference on Decision and Control. pp. 4732–4737, San Diego, CA (1997)
8. Hermandez, J.I., Kuo, C.Y: Steering control of automated vehicles using absolute positioning GPS and magnetic markers. IEEE. Trans. Veh. Technol. **52**(1), 150–161 (2003)
9. Herpin, J., Fekih, A., Golconda, S., Lakhotia, A.: Steering control of the autonomous vehicle: CajunBot. AIAA. J. Aerosp. Comput. Inf. Commun. (JACIC) **4**, 1134–1142 (2007)
10. Hingwe, P., Tomizuka, M.: A Variable look-ahead controller for lateral guidance of four wheeled vehicles. In: Proceedings of the American Control Conference. Pennsylvania, pp. 31–35 (1998)
11. Hsu, J.C, Tamizuka, M.: Analysis of vision-based lateral control for automated highway system. Veh. Syst. Dyn. **30**, 345–373 (1998)
12. Jones, W.D.: Building Safer Cars. IEEE. Spectr. **39**(1), 82–85 (2002)
13. Kiencke, U., Nielsen, L.: Automotive Control Systems: for Engine, Driveline, and Vehicle. 2nd edn. Springer, Berlin (2005)
14. Lakhotia, A., Golconda, S., Maida, A., Mejia, P., Puntambekar, A., Seetharaman, G., Wilson, S.: CajunBot: architecture and algorithms. J. Field. Robot. **23**(8), 555–578 (2006)
15. Lee, H., Tomozuka, M.: Coordinated longitudinal and lateral motion control of vehicles for IVHS. ASME. J. Dyn. Syst. Meas. Control **123**, 535–543 (2001)
16. Li, L., Wang, F-W.: Research advances in vehicle motion monitoring and control. Intern. J. Intell. Control **10**(1), 60–76 (2005)
17. Li, L., Song, J., Song, J., Song, J.: New developments and research trends for intelligent vehicles. IEEE. Intell. Syst. **20**(4), 10–14 (2005)
18. Li, L., Wang, F-W.: Advanced Motion Control and Sensing for Intelligent Vehicles. Springer, New York (2007)
19. Maida, A., Golconda, S., Mejia, P., Lakhotia, A.: Subgoal-based local navigation and obstacle-avoidance using a grid-distance field. Intern. J. Veh. Auton. Syst. **4**(2–4), 122–142 (2006)
20. Ozguner, U., Unyelioglu, K.A., Hatipoglu, C.: An analytical study of vehicle steering control. In: Proceeding of the 4th IEEE Conference on Control Applications. Albany, New York, pp. 125–130, September (1995)
21. Rajamani, R.: Vehicle Dynamics and Control. Springer, New York (2006)
22. Segel, M.C: Theoretical prediction and experimental substation of the response of the automobile to steering control. Proc. Automobil. Div. Inst. Mech. Eng. **7**, 310–330 (1956)
23. Trepagnier, P., Nagel, J., Powell, K.M., Koutsougeras, C., Dooner, M.: KAT-5: Robust systems for autonomous vehicle navigation in challenging and unknown terrain. J. Field. Robot. **23**(8), 509–526 (2006)

Vehicle Architecture and Robotic Perception for Autonomous Driving in Urban Environments

Jan Effertz and Jörn Marten Wille

Abstract We present the vehicle architecture, sensor and data fusion concept as well as the vehicle guidance system of the 2007 Urban Challenge vehicle (Caroline) of Team CarOLO (Technische Universität Braunschweig). The vehicle architecture is subdivided into four main functional levels for perception, situation interpretation, motion planning and motion control. The perception approach is based on a hybrid fusion system, combining classical object-based tracking and grid-based data fusion for obstacle detection and roadway analysis, respectively. A variety of sensor types are applied including radar, fixed beam LIDAR, rotating laser scanners and camera systems. For the tracking system, the classical (e.g. rectangular-shaped) object model is replaced by a polygonal free-form approach suitable for arbitrary obstacle description. The target dynamics estimation algorithm is adopted to cope with the complex shape model. For the grid-based data fusion, the drivability of the roadway is described by a Dempster–Shafer probability mass model targeting to classify the environment into *drivable*, *undrivable* or *unknown*. LIDAR-based distance measurements and camera-based colorspace segmentation results are transformed into and fused within the Dempster−Shafer drivability space. Data fusion allows higher robustness in order to provide safe lateral guidance of the vehicle through the roadway. Following the perception layer, situation assessment and motion planning is carried out by an interrupt extended behavior based approach. Depending on the current driving mode (e.g. free driving, parking or u-turns) an optimized trajectory is calculated and realized through the low-level control units. The system described in this article has been profoundly tested and proved it's quality within the preparation

J. Effertz (✉) · J. M. Wille
Institute of Control Engineering, Technical University Braunschweig,
Hans-Sommer-Str. 66, 38106 Braunschweig, Germany
e-mail: j.effertz@janeffertz.de

J. M. Wille
e-mail: wille@ifr.ing.tu-bs.de

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_9, © Springer-Verlag London Limited 2012

and the final competition of the Urban Challenge event, leading to a ranking of Caroline within the top-ten vehicles.

1 Introduction

The realization of fully-autonomous driving in real-traffic scenarios has been fascinating engineers around the world for decades, from the early beginning of automatic vehicle guidance at the Tsukuba laboratories in the 1970s via the German Prometheus projects around 1990 [11] up to the DARPA Grand Challenges in 2004, 2005 [22, 24] and DARPA Urban Challenge in 2007 [20, 23, 25]. As a premiere, the DARPA Urban Challenge was requiring mixed autonomous and human-controlled traffic in an urban environment, only secured by a remote-controlled emergency stop system [8, 9].

These conditions added new challenges for reliability and long-term robustness of the designed robot vehicles as well as calling for new approaches in environment modeling, allowing safe navigation through complex urban terrain [13].

This contribution reflects the robotic vehicle Caroline, created by Team CarOLO of the Technical University of Braunschweig (Germany), which successfully qualified for the DARPA final race event and finished the competition among the ten best vehicle platforms.

2 Vehicle Architecture

The development of an autonomous driving vehicle is boosted by contributions from the interdisciplinary fields of information science, electrical and control engineering, sensor systems and environment modeling as well as localization and navigation technique. Therefore, the development of a modular vehicle architecture that provided decoupling of vehicle subsystems was key to a successful implementation of the overall functionality. Based on the example of human driving, the task of autonomous driving can be separated into four main functional levels, which satisfy the following questions:

1. *Perception level:* What is happening around me and where am I located?
2. *Interpretation level:* Which effect does the environment have on myself and the necessary decisions to reach my objectives?
3. *Planning level:* Which possibilities of reaction do I have and how can I use these most effectively in order to accomplish that objectives?
4. *Action level:* What is the right way to use the available actors in order to carry out the desired action?

The translation of these four questions into an evolving overall vehicle architecture is depicted in Fig. 1 [13]. Within the perception level, measurement data of

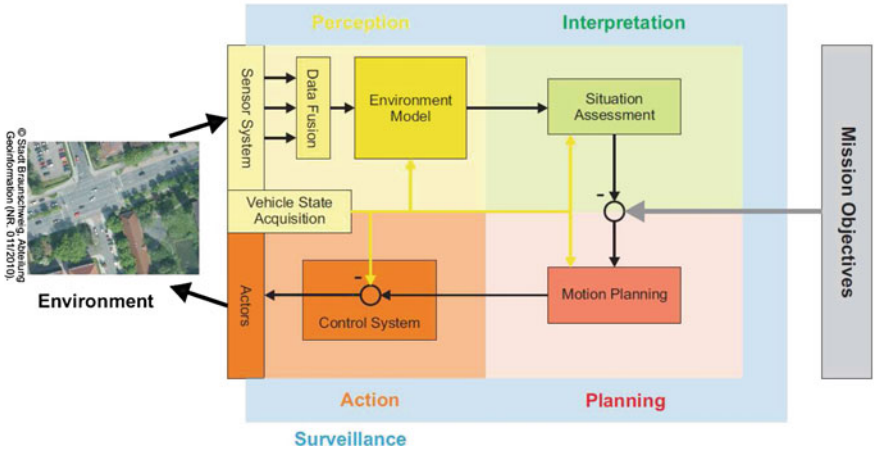


Fig. 1 Overall vehicle architecture

various sensor systems is acquired, fused and converted into a central artificial world model. Based on this model, situation assessment is carried out in the interpretation level. For this purpose, context knowledge (e.g. awareness of an intersection situation) has to be created and the perceived environment components (e.g. detected vehicles) are analyzed within that context. Within the planning level, this situation understanding is aligned with mission objectives in order to deduct a correct vehicle reaction which then is converted into a desired trajectory. Finally, this trajectory is transferred to the action level that feeds the vehicle's low-level motion controllers and creates the correct actor inputs (steering, brake, throttle).

From the perspective of control engineering, the autonomous vehicle can therefore be regarded as a cascade control structure. The outer control loop is closed via the vehicle's environment and provided mission objectives whereas the inner control loop is closed via the vehicle's position, speed and orientation (vehicle state vector) and it's desired trajectory. The vehicle state vector itself and it's precise acquisition is key to the overall system and is taken into account by each of the four functional levels.

With the lack of human supervision or intervention within a fully-autonomous vehicle system, a fifth level of functionality is necessary for surveillance and recovery purpose. It's task is to continuously monitor each of the four main functional levels, to recognize and automatically fix occurring error states and to bring the vehicle to a safe stop in case of emergency.

2.1 Vehicle Platform and Sensor Systems

A standard-build Volkswagen Passat station wagon has been used as Caroline's vehicle platform. Due to Volkswagen's advances in the field of innovative driver

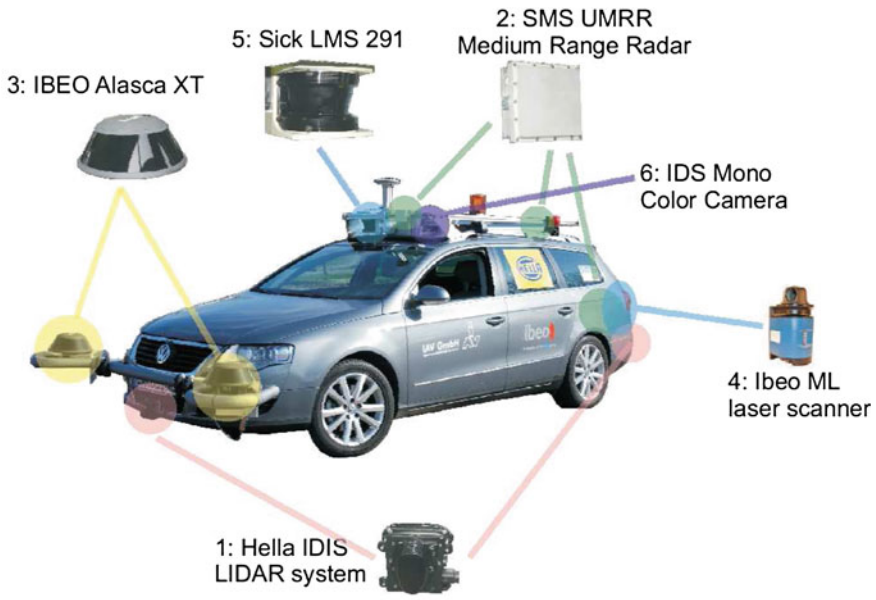


Fig. 2 Vehicle platform and sensor system

assistance systems, the Passat brings along most of the necessary actors, such as the electrical power steering as well as digitally-controllable throttle and brake systems, which can be accessed from the onboard CAN-communication network without additional changes to the vehicle platform itself.

A variety of sensor types originating from the field of driver assistance systems and industrial automation have been chosen to provide detection of static and dynamic obstacles in the vehicle's surrounding, as depicted in Fig. 2.

1. A stationary beam LIDAR sensor has been placed in the front and rear section of the vehicle, providing a detection range of approximately 200 m with an opening angle of 12°. The unit is equipped with an internal pre-processing stage and thus delivers its readings in an object-oriented fashion, providing target distance, target width and relative target velocity with respect to the car's fixed sensor coordinate frame.
2. 24-GHz radar sensors have been added to the front, rear, rear left and right side of the vehicle. While the center front and rear sensors provide a detection range of approximately 150 m with an opening angle of 40°, the rear right and left sensors operate as blind-spot detectors with a range of 15 m and an opening angle of 140° due to their specific antenna structure. The front sensor acts as a stand-alone unit delivering object-oriented target data (position and velocity) through its assigned external control unit (ECU). The three radar sensors in the rear section are operated as a combined sensor cluster using an additional ECU, providing object-oriented target data in the same fashion as the front system.

From the post-processing fusion system perspective, the three rear sensors can therefore be regarded as one unit.

3. Two Ibeo ALASCA XT laser scanners have been installed in the vehicle's front section, each providing an opening angle of 240° with a detection range of approximately 60 m. The raw measurement data of both front laser scanners is preprocessed and fused on their assigned ECU, delivering complex object-oriented target descriptions consisting of target contour information, target velocity and additional classification information.
4. One Ibeo ML laser scanner has been added to the rear side, providing similar detection capabilities like the two front sensors, with a reduced opening angle of 180° due to its mounting position. All Ibeo sensors are based on a four-plane scanning principle with a vertical opening angle of 3.2° between the top and bottom scan plane. Due to this opening angle, smaller pitch movements of the vehicle can be covered.
5. Two Sick LMS-291 laser scanners have been mounted to the vehicle's front-roof section. The scanners are based on a single-plane technology. They have been set up to measure the vertical terrain profile in 10 and 20 m, respectively. The view angle has been limited to 120° by software means.
6. A USB-based color mono camera has been placed onto the vehicle's front roof section, covering an opening angle of approximately 60°.

In order to provide precise localization and ego-state estimation, a combination of satellite and inertial navigation is used.

2.2 Perception Level

The sensor data is post-processed by a hybrid world model, differentiating between distinct *obstacles* (either dynamic or static) and a general *drivability* of the road surface. The former are represented by an object-based tracking system [12] capable of describing arbitrary object shapes by using a freeform polygonal approach, the latter is modeled by a grid-based world model, separating the drivable roadway from non-drivable terrain such as curbs and unpaved road berms.

The sensor system described above reflects a hybrid post-processing scheme. While sensors 1 through 4 in Fig. 2 are delivering their data in an object-oriented fashion, and are therefore treated within the system's object tracking and data fusion stage, the last two described sensors are evaluated and fused based on their raw measurement data in the grid-based subsection. A detailed overview of the object- and grid-based subsystems is given in Sects. 3 and 4 (Fig. 3).

2.3 Interpretation and Planning Level

In order to carry out situation assessment and motion planning, an extended behavior-based decision concept is used. The final vehicle behavior is composed

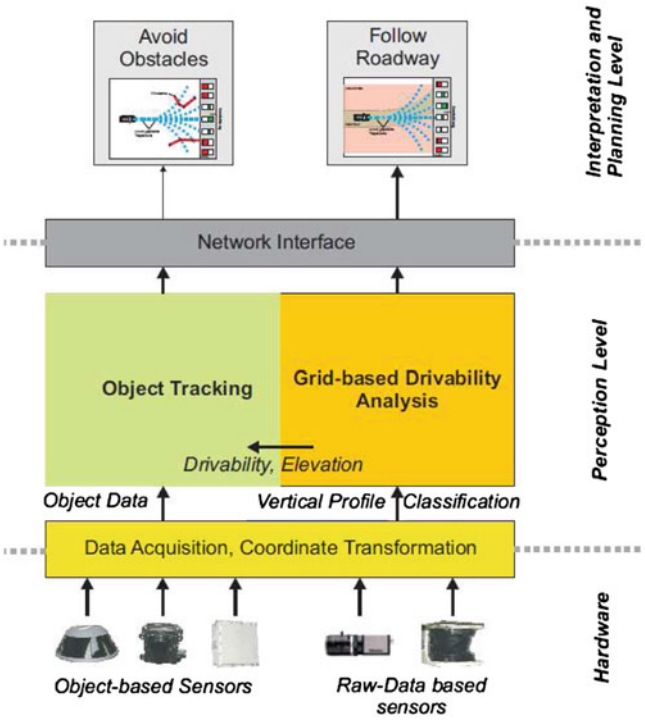


Fig. 3 Hybrid perception system consisting of object-based obstacle tracking and grid-based drivability analysis

out of three main sub-behaviors, taking into account both the hybrid perception model as well as given GPS-based map-material of the urban environment as provided for the Urban Challenge contest. Each behavior model is based on Rosenblatt's DAMN-Approach [21].

Accordingly, motion planning starts recursively from the current vehicle position and is added to previously planned trajectory segments. In each planning step, a fixed number of possible succeeding trajectory segments are evaluated. These are based on arcs with varying curvature starting with the minimum and ending with the maximum drivable curvature that is determined by the vehicle kinematics (Fig. 4a).

Each possible trajectory segment is then voted on by the perception model(s) and given map-material, respectively. As a result, trajectories leading toward or ending in obstacles detected by the object tracking system are voted negative (or even forbidden). Trajectories leading toward drivable roadway described by the grid based section are voted positive. Finally, trajectories leading towards the next desired waypoint extracted from the GPS-based map-material are voted positive as well. As illustrated in Fig. 4b, the three separate sub-behavior votes are then

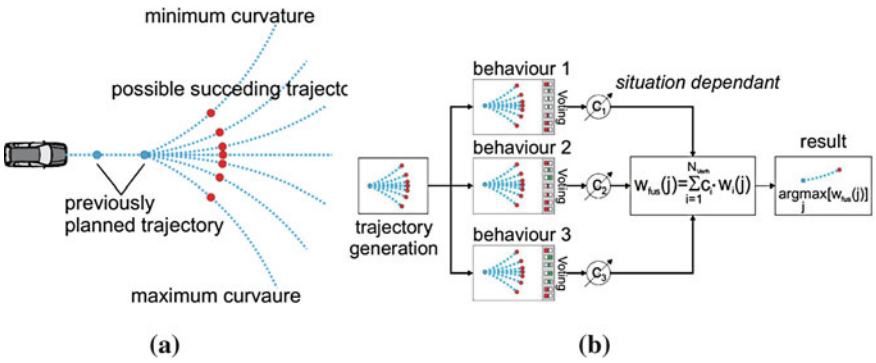


Fig. 4 Behavior-based motion planning: **a** DAMN-based trajectory generation, **b** combination of sub-behaviors

combined into a final vote, taking into account a situation-dependent weight for each sub-behavior. In a similar way, the desired vehicle speed is chosen.

Based on the described strategy, the vehicle would be moving at all times avoiding static and dynamic obstacles and heading toward waypoints described within the mission objectives. In order to be able to obey traffic rules, an interrupt-driven state machine is added to the behavior generation. An interrupt handler is monitoring the current driving situation based on the perception system and given map-material. If a special situation (e.g. handling an intersection) occurs, a specialized interrupt service routine takes over vehicle control by influencing the behavior weighting or directly commanding specialized trajectories, such as parking and u-turns. As a positive effect of this architecture depicted in Fig. 5, more complex functionality of the robot can be added step-by-step, with independent state machines not affecting each other or the general driving behavior in normal traffic.

3 Object-Tracking System

For tracking of dynamic and static obstacles in the vehicle's surroundings, a freeform object model is used based on a polygonal approach to describe arbitrary object contours. Each object is described by

- a number of N_p contour-points $\mathbf{p}_i$ located at the objects outline,
- a state vector $\mathbf{x} = \left[x_c \; y_c \; v \; \phi \; a \; \dot{\phi} \right]^T$ and
- the state covariance matrix $\mathbf{P}$ modeling the state estimation certainty, with x_c and y_c being the motion-origin of all contour points $\mathbf{p}_i$, v object velocity, a object acceleration, ϕ course angle and $\dot{\phi}$ course angle velocity as depicted in Fig. 6. A rigid object outline is assumed.

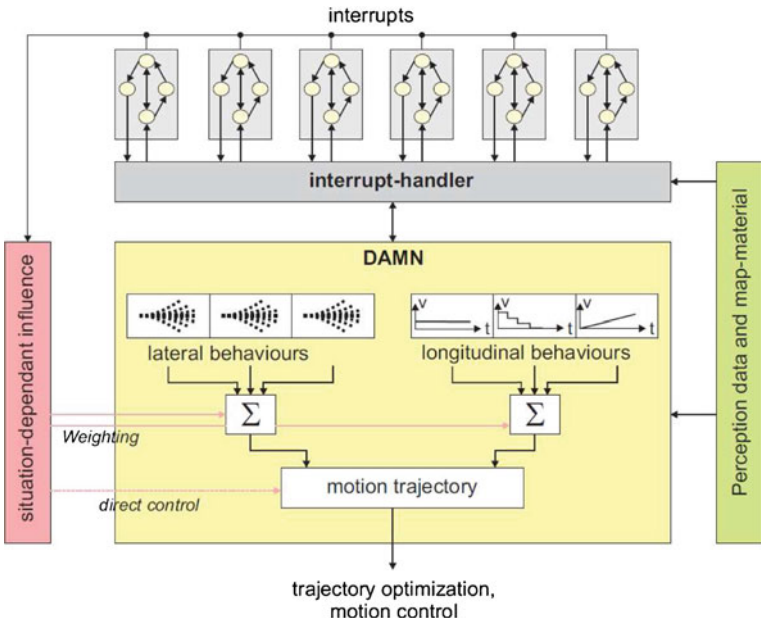


Fig. 5 Behavior-based motion planning extended by interrupt-based situation analysis

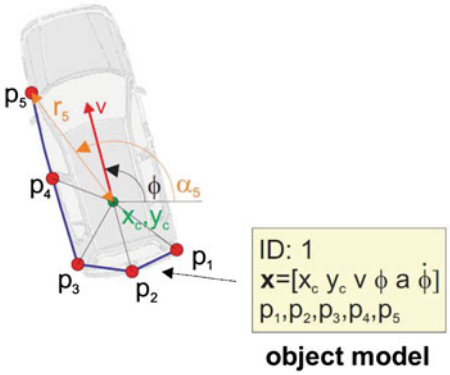


Fig. 6 Freeform object model used in the tracking system

The vertical component is neglected since none of the connected sensor systems is capable of delivering target height information.

3.1 Movement Model

State-vector composition is deducted from a constant-turn Ackerman steering model [18]. The model has been extended by an acceleration term in order to allow

changes of target velocity and can be regarded as a good match for the majority of existing dynamic targets in real traffic scenarios. Based on that definition, the state transition function can be determined as follows:

$$\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{f}(\mathbf{x}_k) \Rightarrow \\
x_{c_{k+1}} &= x_{c_k} + \frac{v_k + \frac{1}{2}a_k \cdot \Delta t}{\dot{\phi}_k} \cdot \Big(\sin(\phi_k + \dot{\phi}_k \cdot \Delta t) - \sin\phi_k\Big), \\
y_{c_{k+1}} &= y_{c_k} + \frac{v_k + \frac{1}{2}a_k \cdot \Delta t}{\dot{\phi}_k} \cdot \Big(\cos\phi_k - \cos(\phi_k + \dot{\phi}_k \cdot \Delta t)\Big), \\
v_{k+1} &= v_k + a \cdot \Delta t, \\
\phi_{k+1} &= \phi_k + \dot{\phi}_k \cdot \Delta t, \\
a_{k+1} &= a_k \quad \text{and} \\
\dot{\phi}_{k+1} &= \dot{\phi}_k.
\end{aligned} \tag{1}$$

The position of each contour point is subject to further rotation around the center of movement x_c and y_c and can be determined by

$$\begin{aligned}
x_{i_{k+1}} &= x_{i_k} + \frac{v_k + \frac{1}{2}a_k \cdot \Delta t}{\dot{\phi}_k} \cdot \Big(\sin(\phi_k + \dot{\phi}_k \cdot \Delta t) - \sin\phi_k\Big) \\
&\quad + r_{i_k} \cdot \Big(\cos(\alpha_{i_k} + \dot{\phi}_k \cdot \Delta t) - \cos\alpha_{i_k}\Big) \quad \text{and} \\
y_{i_{k+1}} &= y_{i_k} + \frac{v_k + \frac{1}{2}a_k \cdot \Delta t}{\dot{\phi}_k} \cdot \Big(\cos\phi_k - \cos(\phi_k + \dot{\phi}_k \cdot \Delta t)\Big) \\
&\quad + r_{i_k} \cdot \Big(\sin(\alpha_{i_k} + \dot{\phi}_k \cdot \Delta t) - \sin\alpha_{i_k}\Big).
\end{aligned} \tag{2}$$

Since the true center of motion is unknown and not necessarily observable, the centroid x_c, y_c of contour points $\mathbf{p}_i$ is considered the center of movement as an approximation. This approximation can be deducted easily as Least-Squares fit of two translated point-clouds without additional rotation and therefore holds for smaller turn rates $\dot{\phi}$ which are likely to be encountered in real traffic due to vehicle dynamics and urban driving speeds.

3.2 Input Data and Signal Flow

Measurement data is delivered by the RADAR and LIDAR systems described in Sect. 1. The measurement information of each sensor is abstracted into a general measurement vector, which is described through

$$\mathbf{y} = \begin{pmatrix} x^{1..m} \\ y^{1..m} \\ v_x \\ v_y \end{pmatrix}, \tag{3}$$

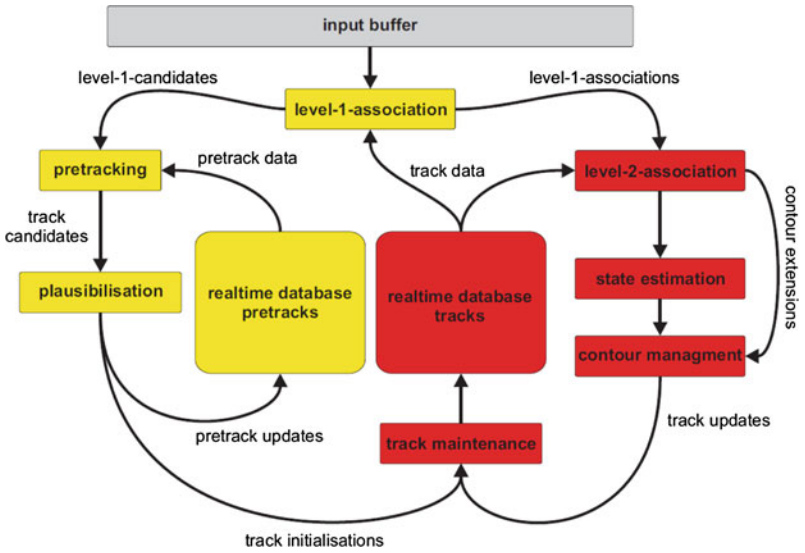


Fig. 7 Tracking-system architecture

with $x^{1..m}$ and $y^{1..m}$ being the m measured contour point x-and y-coordinates with respect to the global earth-fixed reference frame, and v_x as well as v_y being the corresponding object's velocity vector. Depending on the sensor type, some parts of Eq. 3 will remain empty:

- The rotating laser scanners are capable of delivering up to 32 contour points in a common velocity vector, the latter as a result of sensor-internal pretracking.
- The fixed beam LIDAR sensors deliver only the position of the right and left border of any detected object. Due to the underlying measurement principle (time of flight), they cannot directly measure a target's velocity and will therefore rely on differentiating the object's position from within their own pretracking stage—which often results in poor velocity information.
- The radar sensors are only capable of delivering a single point of reflection, whose position on the target is not precisely known and can be randomly distributed. An advantage to the laser-based sensor is that the relative radial target velocity can be precisely measured by evaluating the signal's Doppler shift.

Furthermore, the sensor data is transformed into a global, earth fixed reference frame for all further post processing. This way, static obstacles (which are the majority of all targets) will carry zero velocity, which eases state estimation later on. After data acquisition, the incoming object data is piped into the actual tracking system consisting of two main circuits as depicted in Fig. 7.

3.3 Data Association

To maintain data association incoming sensor readings are first buffered and ordered by their measurement timestamps in order to prevent out-of-sequence tracking situations [2]. Secondly, the incoming measurement vectors are subject to two levels of data association.

- Within the first level of association, the correct dependence between existing track objects and incoming measurement vectors is evaluated. Depending on the type of measurement vector and track-model content, point-to-point, point to polygon or polygon to polygon-comparisons are evaluated and expressed by a scalar matching weight $d_1(i,j)$, expressing similarity between track i and measurement j. Gating of the similarity term $d_1(i,j)$ prevents unlikely matches. An association cost matrix,

$$\mathbf{D}_1 = \begin{bmatrix} d_1(1,1) & \dots & d_1(1,M) \\ & \dots & \\ d_1(N,1) & \dots & d_1(N,M) \end{bmatrix}, \tag{4}$$

  is subsequently created storing similarities for each possible pairing between N existing track objects and M incoming measurement vectors. $\mathbf{D}_1$ can easily be optimized using standard approaches like the Hungarian or Nearest Neighbor algorithm [3, 5], leading to an association matrix

$$\Omega_1 = \begin{bmatrix} \omega_{1,1} & \cdots & \omega_{1,M} \\ \cdots & \cdots & \cdots \\ \omega_{N,1} & \cdots & \omega_{N,M} \end{bmatrix}, \quad \text{with} \quad \begin{matrix} \sum_{i=1}^{N} \omega_{i,k} \leq 1 \; \forall \; k \\ \sum_{j=1}^{M} \omega_{k,j} \leq 1 \; \forall \; k \end{matrix} \quad \text{and} \quad \omega_{i,j} \in [0,1]. \tag{5}$$

- Within the second level of association, positive matches from the first level are more finely associated in order to assign each contour point within the measurement vector y to the corresponding contour point within the track definition. An association matrix $\mathbf{D}_2$ is then created,

$$\mathbf{D}_2 = \begin{bmatrix} d_2(1,1) & \dots & d_2(1,M_p) \\ & \dots & \\ d_2(N_p,1) & \dots & d_2(N_p,M_p) \end{bmatrix}, \tag{6}$$

  storing a similarity value $d_2(k,l)$ for each possible point pairings of the N_p track contour points and M_p measurement vector contour points. In order to calculate $d_2(k,l)$, the Euclidian point distance as well as the measurement and track covariance is evaluated. The same optimization algorithm like in level-1-association can be applied to determine the best point-to-point matches leading to an association matrix Ω_2. Unassociated points within the measurement vector are candidates for contour extensions within the track.

  Figure 8a, b illustrates the two association steps.

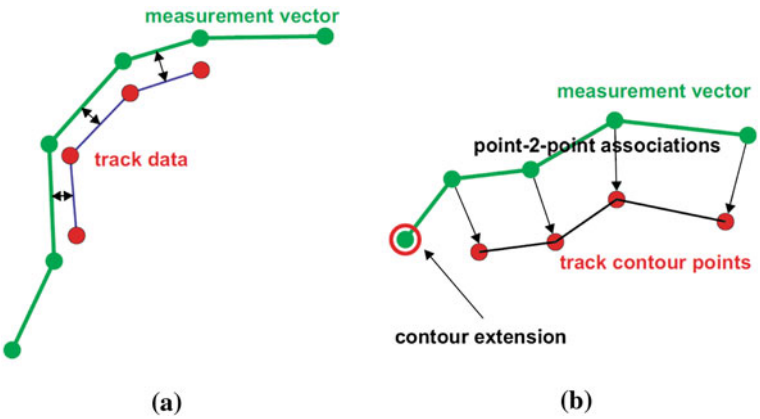


Fig. 8 Two-step data association for object tracking: **a** level-1-association, **b** level-2-association

3.4 Pretracking

While it would be possible to simply initiate a new track for all measurements that could not be associated with any existing track during level-1-association, this would lead to an abundance of false alarms and decrease the tracking system's performance significantly, as the computational load would rise due to the inefficiency and false alarms would compete against the correct tracks for incoming sensor readings. In Caroline's fusion system, a database of track candidates is therefore filled with incoming sensor readings that could not be associated. A simple tracking based on the closest detected point within each measurement vector is carried out in order to justify track candidates prior to their activation. Only if a track candidate has been acknowledged more than once by one or more sensors, will it be activated and moved to the main tracking memory. Additionally, information from the grid-based drivability analysis is used to cancel out unwanted ground reflections from the laser scanners.

3.5 Tracking

An extended Kalman filter [5] is used to carry out state estimation within the main tracking stage. Since the standard approach of Kalman-filtering would normally only handle state vectors consisting of a single position, velocity or further derivatives and thus not take into account the polygonal object model, the algorithm had to be extended. By approximating the centroid of all associated contour points as the center of object movement (refer to Sect. 1) and expressing measurement noise with respect to that point, the Kalman update cycle can be rewritten as:

$$\mathbf{x}^k(v+1|v) = f(\mathbf{x}^k(v|v)), \tag{7}$$

$$\mathbf{P}(v+1|v) = \mathbf{F}^T \cdot \mathbf{P}(v|v) \cdot \mathbf{F} + \mathbf{Q}, \tag{8}$$

$$\mathbf{s}^{k,l}(v+1) = \mathbf{y}^l(v+1) - h(\mathbf{x}^k(v+1|v)), \tag{9}$$

$$\mathbf{S}(v+1) = \mathbf{H} \cdot \mathbf{P}(v+1|v) \cdot \mathbf{H}^T + \mathbf{R}, \tag{10}$$

$$\mathbf{K}(v+1) = \mathbf{P}(v+1|v) \cdot \mathbf{H}^T \cdot \mathbf{S}^{-1}(v+1) \quad \text{and} \tag{11}$$

$$\mathbf{r}^{k,l}(v+1) = \mathbf{K}(v+1) \cdot \mathbf{s}^{k,l}(v+1), \tag{12}$$

with $\mathbf{x}^k$ being the state vector with respect to tracked contour point k and $\mathbf{y}^l$ the measurement vector with respect to measured contour point l, $f(\mathbf{x})$ the nonlinear system transfer function and $\mathbf{F}$ it's associated Jacobian, $\mathbf{P}$ the common state noise covariance matrix, $\mathbf{Q}$ the system noise covariance matrix, $\mathbf{s}^{k,l}$ the innovation vector with respect to tracked and measured contour point k and l, $h(\mathbf{x})$ the system output function and $\mathbf{H}$ it's associated Jacobian, $\mathbf{S}$ the innovation noise covariance matrix, $\mathbf{R}$ the measurement noise covariance matrix, $\mathbf{K}$ the resulting Kalman gain and $\mathbf{r}^{k,l}$ the correction vector for tracked contour point k updated by measured contour point l. While the corrected contour point positions can be immediately retrieved by adding the first two components of $\mathbf{r}^{k,l}$ to the position values, the update for common velocity, course angle, acceleration and course angle velocity can be acquired by calculating the mean of $\mathbf{r}^{k,l}$ over all N contour point associations:

$$\mathbf{r}^{\text{mean}} = \frac{1}{N} \sum_{\forall k,l \exists \Omega_2[k,l] \equiv 1} \mathbf{r}^{k,l}(v+1). \tag{13}$$

4 Grid-Based Drivability Analysis

The grid-based drivability analysis aims to classify discrete subsets (grid cells) of the environment into drivable and undrivable terrain. The environment model is given by a grid map $\mathbf{Z}$ consisting of fixed number of grid cells $z(i, j)$ where the cell indices i, j are unambiguously determining a cell's position in the real world. The underlying coordinate system for the map is a world fixed with an arbitrary (but known) origin expressed in real-world coordinates x^{ref}, y^{ref} (e.g. a north-east well as the j and y axes are aligned. Given any coordinates x, y in real-world coordinates, the corresponding discrete coordinates i, j can be calculated as

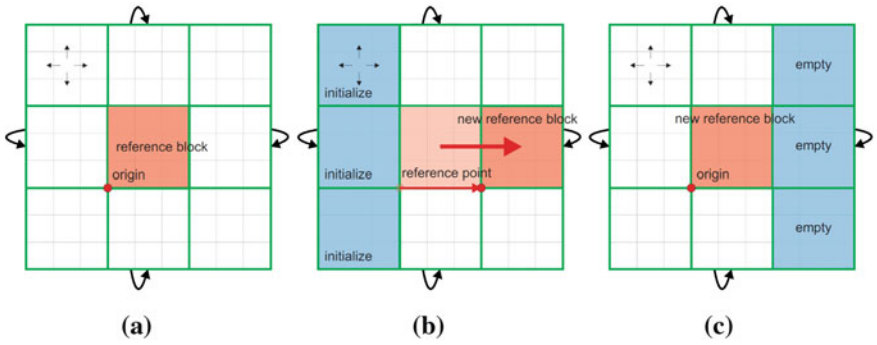


Fig. 9 Subblock-based cyclic list for efficient map implementation: **a** Initial condition, **b** transition, **c** final condition

$$\begin{aligned} i &= \text{round}\,\frac{x - x^{\text{ref}}}{\Delta x} \\ j &= \text{round}\,\frac{y - y^{\text{ref}}}{\Delta y} \end{aligned} \tag{14}$$

Cell resolution Δx, Δy is constant for the implementation discussed here. The content of each grid cell is arbitrary and will be discussed later within this section.

Since the grid map is world fixed and the vehicle moves through the map, it is not feasible to cover an unlimited area in real-world coordinates. The map-boundaries are therefore limited and as the vehicle moves through the real world, new space is allocated at the front end while old space goes out of scope and has to be deallocated at the rear. For an efficient implementation [17], the map has been set up as a cyclic chained list with separated sub-blocks. The origin of the map is therefore influenced by the vehicle's position and moved from sub-block to sub-block. In parallel, the content of new *appearing* blocks at the virtual horizon is cleared as depicted in Fig. 9a up to 9c.

In order to model drivability within the grid map, an approach similar to classical occupancy grids [14] is chosen. Instead of a using Bayesian probability modeling for drivability, a Dempster–Shafer [10] mass distribution is assigned to each grid cell $z_(i,j)$, leading to a probability mass distribution

$$\begin{aligned} m_{z(i,j)}[B] &= m[\text{drivable}], \\ m_{z(i,j)}[U] &= m[\text{undrivable}] \quad \text{and} \\ m_{z(i,j)}[K] &= m[\text{unknown / no information}]. \end{aligned} \tag{15}$$

An advantage of the Dempster–Shafer approach, in comparison to Bayes' occupancy grids, is that it is possible to explicitly model unknown areas in the grid map which have not been filled with measurement information so far and distinguish them from grid cells with conflicting sensor data [16, 26]. Coded into r–g–b color

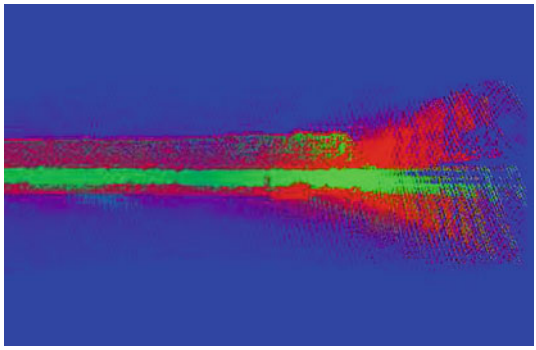


Fig. 10 Example drivability analysis of a straight roadway separated by undrivable road berms on both sides

values for undrivable, drivable and unknown, the world model depicted in Fig. 10 is created.

In order to carry out the drivability analysis, separate layers of information are established in each grid cell:

- A height layer storing the vertical profile of the environment,
- A gradient layer storing local height changes,
- A color similarity layer storing results from a color camera-based roadway analysis,
- A Dempster–Shafer integration layer storing drivability masses deducted from the gradient information,
- A Dempster–Shafer integration layer storing drivability masses deducted from the color similarity layer and,
- A fusion layer storing fused drivability masses from the gradient and color similarity information.

The signal flow between each layer is depicted in Fig. 11 and will be discussed in detail in the following sections.

4.1 LIDAR-Based Generation of Height Profile

In order to create a height profile of the environment, the distance measurements of both Sick scanners (refer to Sect. 1) are evaluated. A ray is cast through the map from the sensor origin to the end-point of each distance value, updating each grid cell that lies between the start and end point. At the end point, the height information corresponding with the distance reading is stored in the grid cell. Any old height values within the grid map that exceed the actual cast ray are reset, since based on the LIDAR-technique, the sensor is not able to see *through* existing obstacles. The procedure is depicted in Fig. 12. As the vehicle moves through the environment, the height layer of the grid map is filled with information incrementally.

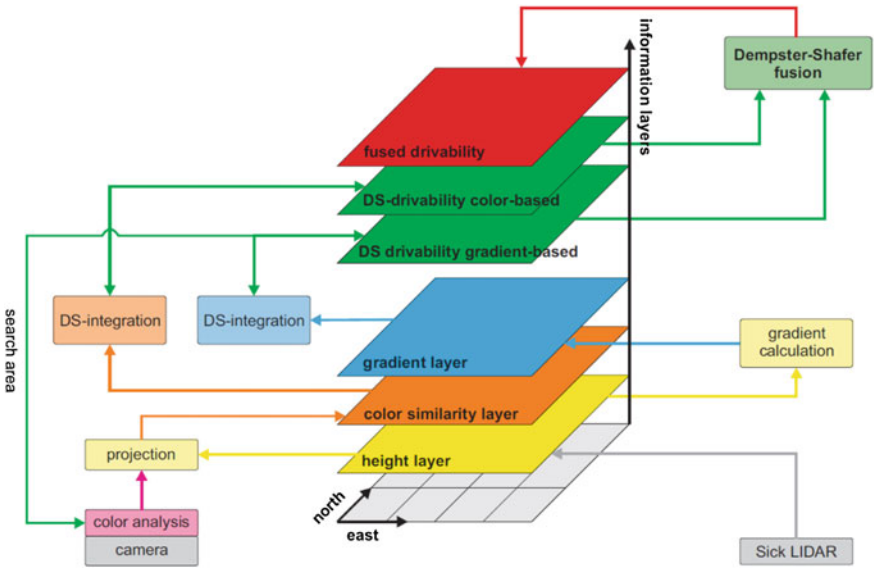


Fig. 11 Data layers and signal flow of the grid-based drivability analysis

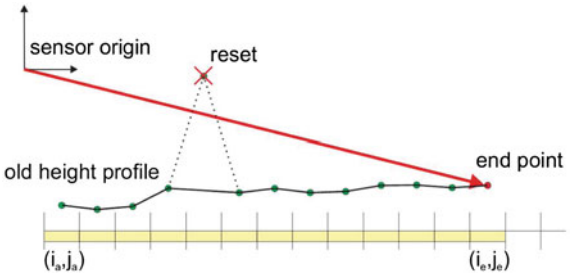


Fig. 12 Raycasting of LIDAR-readings into the grid map in order to create a vertical profile of the environment

For the raycasting itself, a 2D-variant of the Bresenham algorithm [6] is executed. With a given start point x_1, y_1 and target point x_2, y_2 in discrete coordinates, the Bresenham algorithm continues basically as follows,

1. Compute $\Delta x = x_1 - x_0$.
2. Compute $\Delta y = y_1 - y_0$. We hereby assume that $\Delta x \geq \Delta y$ and we want to trace a line through the first quadrant, otherwise the coordinates must be swapped.
3. Initialize an error variable E to $\frac{\Delta x}{2}$.
4. Start with x_0 and increase up to x_1 by step size 1.
5. For each step, decrease E by Δy.
6. If $E \leq 0$, increase E by Δx and increase y by one.
7. Stop, if reached x_1, otherwise continue with step 5.

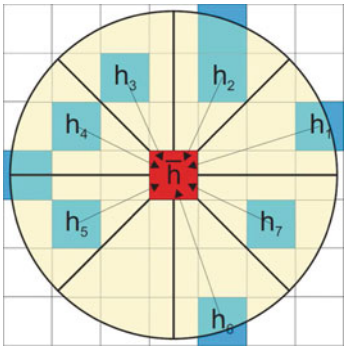


Fig. 13 Expand-algorithm

The lines are traced similar to the functionality of a plotter, which is the origin of that algorithm.

4.2 Interpolation and Gradient Analysis

In order to prepare gradient analysis, the height profile is interpolated to compensate for missing height values, which can occur if the vehicle movement is fast enough so that the raycasting skips cells from one measurement cycle to the next. Interpolation is carried out both locally and later on globally. Local interpolation is based on a simple Expand-algorithm, filling *holes* in the map with the mean values of neighboring grid cells (see Fig. 13).

For the global interpolation, a ground plane

$$\mathbf{n} \cdot \mathbf{p}_i - d = 0 \tag{16}$$

with normal vector $\mathbf{n}$ and distance term d is estimated from existing height values in the grid map. The estimation is performed by a weighted Least-Squares approach, minimizing the residuum

$$R(n_x, n_y, n_z, d) = \sum_{i=1}^{N} w_i \cdot e^T \cdot e = \sum_{i=1}^{N} w_i \cdot (n_x x_i + n_y y_i + n_z z_i - d)^2. \tag{17}$$

The weights w_i are taken from the Dempster–Shafer drivability stored in the grid map from a previous measurement cycle in order to suppress existing obstacles from the ground-plane calculation. As a result, the distance value d can be calculated by

$$d = \frac{1}{\sum_{i=1}^{N} w_i} \begin{pmatrix} \sum_{i=1}^{N} w_i x_i \\ \sum_{i=1}^{N} w_i y_i \\ \sum_{i=1}^{N} w_i z_i \end{pmatrix} \cdot \mathbf{n}. \tag{18}$$

The normal vector $\mathbf{n}$ can be resolved as a solution of the Eigenvalues of matrix $\mathbf{P}$ [1], with

$$\mathbf{P}=\begin{bmatrix} \sum_{i=1}^{N} w_i(x_i-x_c)^2 & \sum_{i=1}^{N} w_i(x_i-x_c)(y_i-y_c) & \sum_{i=1}^{N} w_i(x_i-x_c)(z_i-z_c) \\ \sum_{i=1}^{N} w_i(x_i-x_c)(y_i-y_c) & \sum_{i=1}^{N} w_i(y_i-y_c)^2 & \sum_{i=1}^{N} w_i(y_i-y_c)(z_i-z_c) \\ \sum_{i=1}^{N} w_i(x_i-x_c)(z_i-z_c) & \sum_{i=1}^{N} w_i(y_i-y_c)(z_i-z_c) & \sum_{i=1}^{N} w_i(z_i-z_c)^2 \end{bmatrix}. \tag{19}$$

One gets

$$[\mathbf{P}-\lambda\cdot\mathbf{I}]\cdot\mathbf{n}=0, \tag{20}$$

which can easily be solved by common linear algebra algorithm toolboxes.

In order to calculate the local height gradient of the grid map, a convolution with the Sobel-kernel $\mathbf{O}_{\text{Sobel}}$ [15] is applied for both coordinate axes, with

$$\mathbf{O}_{\text{Sobel}_x}=\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{O}_{\text{Sobel}_y}=\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{21}$$

The results of both convolutions are added and stored within the gradient layer $g_{z(i,j)}$ of the grid map, with

$$g_{z(i,j)}=O_{\text{Sobel}_x}\otimes h_{z(i,j)}+O_{\text{Sobel}_y}\otimes h_{z(i,j)}. \tag{22}$$

4.3 Sensor Model for Gradient Analysis

In order to deduct a Dempster–Shafer drivability from the gradient data, a sensor model has to be developed which transforms gradient values into input drivability masses m^{grad} that can be integrated within the gradient-based drivability layer. Accordingly, a sensor model has been defined converting high gradients into a mass distribution supporting undrivability and smooth height profiles into a distribution supporting drivability, as high gradients will be caused by undrivable environment elements such as curbs, obstacles or road berms. One gets

$$\begin{aligned} m^{\text{grad}}[B] &= \begin{cases} B_{\max}, & |g_{z(i,j)}|\le g_{\min} \\ 0, & g_{\min}<|g_{z(i,j)}|\le g_{\max} \\ 0, & |g_{z(i,j)}|>g_{\max} \end{cases}, \\ m^{\text{grad}}[U] &= \begin{cases} 0, & |g_{z(i,j)}|\le g_{\min} \\ \frac{U_{\max}}{g_{\max}-g_{\min}}\cdot\left(|g_{z(i,j)}|-g_{\min}\right), & g_{\min}<|g_{z(i,j)}|\le g_{\max} \\ U_{\max}, & |g_{z(i,j)}|>g_{\max} \end{cases} \quad \text{and} \\ m^{\text{grad}}[K] &= 1-m^{\text{grad}}[B]-m^{\text{grad}}[U]. \end{aligned} \tag{23}$$

The shape of the resulting drivability mass distribution is depicted within Fig. 14a, c.

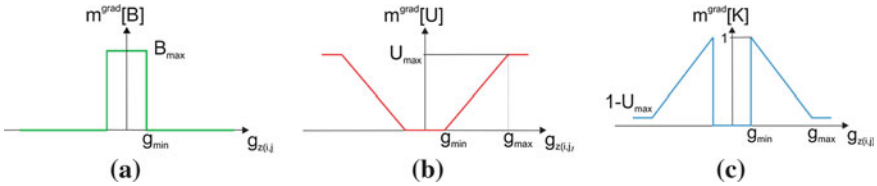


Fig. 14 Gradient-based drivability mass distribution: **a** drivability, **b** undrivability, **c** unknown

Fig. 15 Color-profile-based image analysis: **a** input image and search area, **b** color classification with suppressed areas marked in red

4.4 Color Image-Based Roadway Analysis

As a second source of information for drivability analysis, the color profile of the environment acquired by the mono color camera is evaluated. For this purpose, a search area directly in front of the vehicle is extracted that already supports high drivability. As an assumption, this area is likely to be a drivable roadway. The search area is reprojected into the image plane of the color camera (Fig. 15a) and the corresponding color profile is extracted based on an approach described in [4, 7].

Since the extracted color profile is likely to be characteristic for drivable roadways based on the positioning of the search area, the rest of the camera image is classified based on similarity of each pixel to that color profile. As a result, a scalar value for each pixel ranging from 0 for dissimilarity up to 1 for complete similarity is assigned to each pixel. In order to suppress artifacts originating from shadows or colored lane markings, a set of color filters is applied to mark certain image regions as invalid [red areas in Fig. 15b].

After successful classification, the image is projected onto the grid map and the similarity values of each pixel are stored within the color similarity layer of the map.

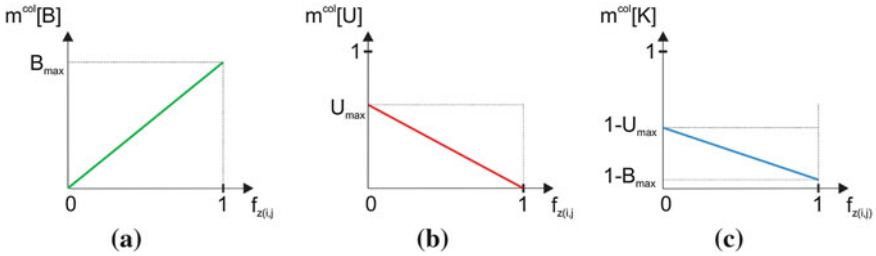


Fig. 16 Color profile-based sensor model: **a** Drivability, **b** Undrivability, **c** Unknown

4.5 Sensor Model for Color Similarities

In order to deduct a drivability mass distribution m^{col} based on the color similarities acquired so far, a sensor model has been created with high values of similarity supporting drivability and vice versa.

$$\begin{aligned} m^{\text{col}}[B] &= B_{\max} \cdot f_{z(i,j)}, \\ m^{\text{col}}[U] &= U_{\max}(1 - f_{z(i,j)}) \quad \text{and} \\ m^{\text{col}}[K] &= 1 - m^{\text{col}}[B] - m^{\text{col}}[U] = 1 - f_{z(i,j)} \cdot (B_{\max} - U_{\max}) - U_{\max}. \end{aligned} \tag{24}$$

The resulting drivability mass distribution is depicted in Fig. 16a, c.

4.6 Data Integration and Data Fusion

Finally, the gradient and color-based drivability mass distributions are integrated over time separately and stored in their respective layers of the grid map. For each grid cell $z(i,j)$, one gets the integrated drivability mass distribution $m^{\text{grad}}_{z(i,j)}$ and $m^{\text{col}}_{z(i,j)}$ by

$$\begin{aligned} m^{\text{grad}}_{z(i,j)}[B] &= \frac{m^{\text{grad}}_{z(i,j)}[B] \cdot m^{\text{grad}}[B] + m^{\text{grad}}_{z(i,j)}[K] \cdot m^{\text{grad}}[B] + m^{\text{grad}}_{z(i,j)}[B] \cdot m^{\text{grad}}[K]}{\eta^{\text{grad}}}, \\ m^{\text{grad}}_{z(i,j)}[U] &= \frac{m^{\text{grad}}_{z(i,j)}[U] \cdot m^{\text{grad}}[U] + m^{\text{grad}}_{z(i,j)}[K] \cdot m^{\text{grad}}[U] + m^{\text{grad}}_{z(i,j)}[U] \cdot m^{\text{grad}}[K]}{\eta^{\text{grad}}} \quad \text{and} \\ m^{\text{grad}}_{z(i,j)}[K] &= 1 - m^{\text{grad}}_{z(i,j)}[B] - m^{\text{grad}}_{z(i,j)}[U] \quad \text{as well as} \end{aligned} \tag{25}$$

$$m^{\text{col}}_{z(i,j)}[B] = \frac{m^{\text{col}}_{z(i,j)}[B] \cdot m^{\text{col}}[B] + m^{\text{col}}_{z(i,j)}[K] \cdot m^{\text{col}}[B] + m^{\text{col}}_{z(i,j)}[B] \cdot m^{\text{col}}[K]}{\eta^{\text{col}}},$$

$$m^{\text{col}}_{z(i,j)}[U] = \frac{m^{\text{col}}_{z(i,j)}[U] \cdot m^{\text{col}}[U] + m^{\text{col}}_{z(i,j)}[K] \cdot m^{\text{col}}[U] + m^{\text{col}}_{z(i,j)}[U] \cdot m^{\text{col}}[K]}{\eta^{\text{col}}} \quad \text{and}$$

$$m^{\text{col}}_{z(i,j)}[K] = 1 - m^{\text{col}}_{z(i,j)}[B] - m^{\text{col}}_{z(i,j)}[U]. \tag{26}$$

The normalization η in Eqs. (25) and (26) are calculated by

$$\eta^{\text{grad/col}} = 1 - m^{\text{grad/col}}_{z(i,j)}[B] \cdot m^{\text{grad/col}}[U] - m^{\text{grad/col}}_{z(i,j)}[U] \cdot m^{\text{grad/col}}[B]. \tag{27}$$

While it would theoretically be possible to integrate both distributions into the same grid layer, different cycle times of the underlying sensor systems would outbalance the fusion process in favor for the faster sensor system. Therefore, both layers are combined after separate temporal integration in order to get a fused drivability value. This value can be received by applying Dempster's rule of combination one more time. One gets

$$m^{\text{fus}}_{z(i,j)}[B] = \frac{m^{\text{grad}}_{z(i,j)}[B] \cdot m^{\text{col}}_{z(i,j)}[B] + m^{\text{grad}}_{z(i,j)}[K] \cdot m^{\text{col}}_{z(i,j)}[B] + m^{\text{grad}}_{z(i,j)}[B] \cdot m^{\text{col}}_{z(i,j)}[K]}{\eta},$$

$$m^{\text{fus}}_{z(i,j)}[U] = \frac{m^{\text{grad}}_{z(i,j)}[U] \cdot m^{\text{col}}_{z(i,j)}[U] + m^{\text{grad}}_{z(i,j)}[K] \cdot m^{\text{col}}_{z(i,j)}[U] + m^{\text{grad}}_{z(i,j)}[U] \cdot m^{\text{col}}_{z(i,j)}[K]}{\eta} \quad \text{and}$$

$$m^{\text{fus}}_{z(i,j)}[K] = 1 - m^{\text{fus}}_{z(i,j)}[B] - m^{\text{fus}}_{z(i,j)}[U]. \tag{28}$$

as the fused drivability mass distribution. The resulting fused drivabilities as well as the height profile are distributed to the vehicle network in order to be used in the motion planning algorithms as well as in the pretracking stage of the object-based perception subsystem as mentioned in Sect. 3.4.

5 Realizing Complex Autonomous Driving Maneuvers

Having a good vehicle guidance system installed into Caroline, it allows the realization of precise autonomous driving maneuvers in an urban environment. The system to guide the vehicle through the scenarios of the Urban Challenge is separated into a decision unit, a trajectory planning that handles driving maneuvers consecutively and a low-level vehicle control module. The vehicle guidance system is in charge to convert abstract decisions into a smooth driving experience. The remainder of the paper will focus on trajectory planning, low-level control and Caroline's overall vehicle guidance system.

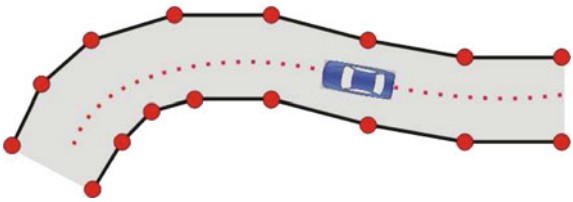


Fig. 17 Interface between decision unit and path planning

5.1 Flexible Interface for Realizing High-Level Decisions

A compact interface was realized to decouple decision unit from final trajectory planning and control system. The structure allows as much freedom as possible for the trajectory planning module but does not contort information substance. As a result, it allows manifold driving strategies to make precise driving maneuvers, whereas two different strategies have been implemented so far. In addition, different control approaches can be easily tested.

The mode 'follow trajectory' is the most commonly used driving mode. The decision unit determines a drivable area, which is bounded through so-called trajectory pearls. The pearls are positioned based on the current environmental conditions, whereas the distance and width of the pearls are adaptable. As a result, those pearls result in a representation of the road edge and a trajectory planner can compute a finally optimized trajectory in between those limits (Fig. 17). For this purpose, an approach is used, which is based on elastic band theory and smoothing splines and performs a dynamically placement of nodes inside the drivable area in realtime.

A pair of pearls does not only contain the edge of the roadway, but also a minimum and maximum speed the vehicle should have at the pearls' position. Next, a speed planning module determines based on this basic conditions a dynamically optimized speed profile. Comfort levels, which are parameters to influence for instance the expected accelerations, open the decision unit the opportunity to affect the driving style of the vehicle.

To realize u-turns, a second driving mode called 'u-turn' is introduced into Caroline's vehicle guidance system. Instead of a list of pearls, the decision unit determines a bounding box, wherein the path planning module realizes a u-turn (Fig. 20). Both modes are illustrated in the following.

5.1.1 Automated-Parking Maneuvers

Vehicles had to pull forward into, and reverse out of a predetermined parking spot in the final event of the DARPA Urban Challenge. The driving mode 'follow trajectory' was used to fulfill these tasks. A list of trajectory pearls is generated to guide Caroline into a parking spot.

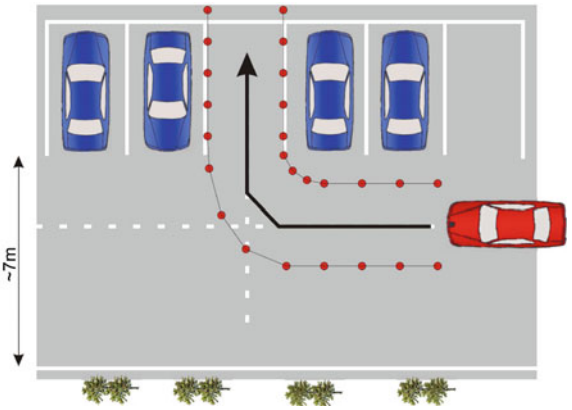


Fig. 18 Parking situation

A state machine, which is based on three major steps, realizes automated parking according to the DARPA requirements:

1. The decision unit guides the vehicle in the proximity of the parking spot.
2. A parking interrupt is started as soon as the condition is fulfilled that parking maneuver can be realized.
3. A multi-stage algorithm guides the vehicle into the desired spot.

The state machine includes two transitions to enter the final state. In favor the vehicle has either been positioned on a line perpendicular or on a straight line into the box (Fig. 18). If the transition condition is met that the state 'enter parkbox' can be performed, a trajectory is aligned to an assistance line that guides the vehicle inside the parking spot. If the box cannot be entered at once, the vehicle is backed up until a successful trajectory is found.

Figure 19 shows a snapshot from the race, where Caroline successfully entered a parking box.

5.1.2 U-Turn Maneuvers

Different situations of the Urban Challenge, as there are blocked roads and dead-end streets, necessitate a multi-stage turn of the vehicle. Those u-turns are realized through an u-turn interface; so that these maneuvers can be finished faster since direct steering commands are used. If a u-turn should be performed, the decision unit sends a bounding box to the path planning module (Fig. 20). The bounding box describes the desired u-turn area, based on static and dynamic obstacles, current street conditions, and lane markings if available. If dynamic obstacles rise into the bounding box during the u-turn procedure, the u-turn maneuver can be interrupted by higher-level decisions at any time.

A u-turn procedure consists of four states. In the state 'enter' the vehicle is brought into the bounding box parallel to its alignment. The two states 'forward

Fig. 19 Snapshot of a parking maneuver during the race

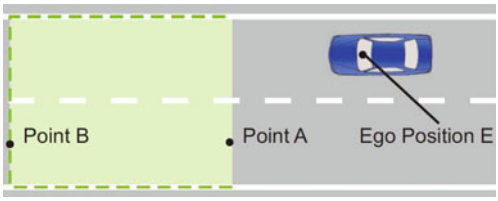


Fig. 20 Bounding box of a u-turn

left' and 'backward right' are consecutively executed to turn the vehicle. As soon as the vehicle is brought into a position that the box can be left with a single maneuver, the state 'finish' to leave the u-turn area is used. Depending on each state steering commands are calculated. Figure 21 visualizes the different states.

A condition is necessary to determine, if a steering angle can be applied that leads the vehicle to the final u-turn position A. Therefore, radius R to provide a curved trajectory is observed:

$$r = \frac{a}{b} \tag{29}$$

$$\begin{aligned} a = {} & B_x(A_y - B_y) - B_y(A_x - B_x) \\ & + E_y(A_x - B_x) - E_x(A_y - B_y) \end{aligned} \tag{30}$$

$$\begin{aligned} b = {} & \left(\cos\left(\alpha - \frac{\pi}{2}\right) - \cos\left(\psi - \frac{\pi}{2}\right)\right)(A_y - B_y) \\ & + \left(\sin\left(\psi - \frac{\pi}{2}\right) - \sin\left(\alpha - \frac{\pi}{2}\right)\right)(A_x - B_x) \end{aligned} \tag{31}$$

Here, ψ is the current orientation of the vehicle, and α the orientation of final u-turn position in global coordinates. Based on Caroline's wheelbase $l = 2.712\,\text{m}$ it has to be checked if the final steering angle is below maximum possible steering angle:

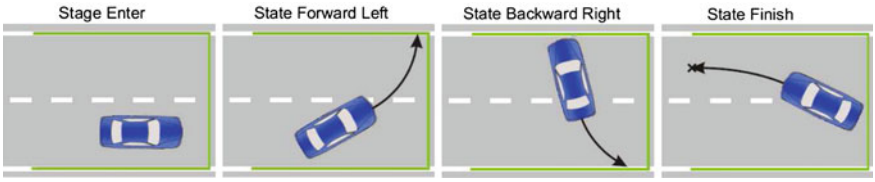


Fig. 21 Different states of the u-turn procedure

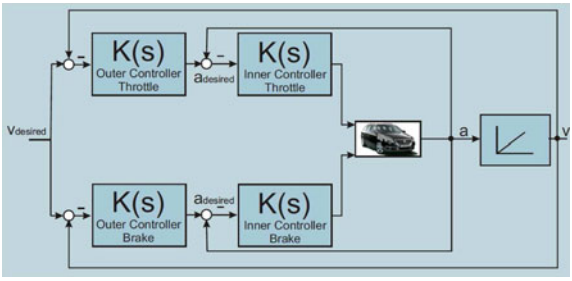


Fig. 22 Block diagram of the longitudinal controller

$$\delta_{\text{finish}} = \arctan\left(\frac{2.712}{r}\right) \leq \delta_{\max} \tag{32}$$

An overview about the different states of the u-turn procedure is given in Fig. 21.

5.2 *Low-Level Control Strategies*

The decisions of an autonomous vehicle are executed by the low-level controllers. While the longitudinal control module is used to follow a speed profile, the lateral controller guides the vehicle on a mathematically described trajectory. Both approaches as installed in Caroline are discussed in the following.

5.2.1 Longitudinal Control Concept

The decision unit sets the limits for vehicle speed. The speed planning module prepares a dynamic optimized speed profile in between these limits that takes the comfort level set by the decision unit into account. The control strategy must finally calculate the braking and accelerator set points in order to maintain a given speed.

Controller Structure: The longitudinal controller consists of an outer and inner loop Fig. 22. The required acceleration, which is needed for a given speed, is determined by the outer loop controller (PI controller). Next, the controller of the inner loop (P controller) calculates throttle and brake commands to track the input

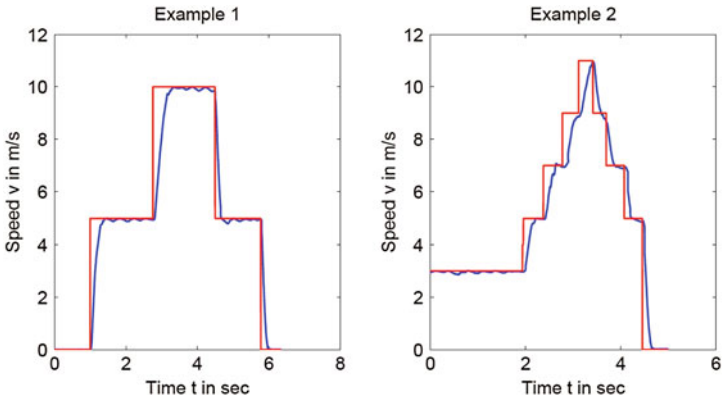


Fig. 23 Longitudinal control performance

of the outer loop. Additionally, an engine map is used for direct feed forward control of the throttle. Acceleration and deceleration operations need to be handled separately with different parameters for each transfer function *K(s)*. A predefined logic combines the controller output to prevent the throttle and brake from being activated at the same time.

Performance of the Longitudinal Controller: In Fig. 23 two different speed profiles are used to illustrate the performance of the longitudinal control strategy. While in the first example, the desired speed is changed in long and large steps, in the second example the speed is changed in shorter and smaller steps.

5.2.2 Lateral Control Concept

In the DARPA Urban Challenge the vehicles had to follow complex trajectories at a typical urban speeds and with a minimum of track error. Also, the driving behavior is supposed to be comfortable. The lateral control strategy which is used in Caroline allows both straight roads and narrow curves to be travelled at high speed.

Controller Structure: For the development of a pilot control, motion behavior of the vehicle is described using the well-known bicycle model [19], which uses therefore three degrees of freedom (x and y position, orientation ψ of the car) while rolling and pitch of the vehicle is neglected. The model assumes that the center of mass is on street level in between front tires. In the lateral controller the steering angle δ handles all degrees of freedom simultaneously. Controller deviations in terms of track angle and track error serve as feedback signals for the controller. They are described mathematically to introduce them into a state space representation of the system. The track error is known as the distance between center of mass of the vehicle and its desired position. The track angle is defined as the difference between the desired and actual orientation of the car. The lateral controller consists of two parallel control loops for track error and track angle

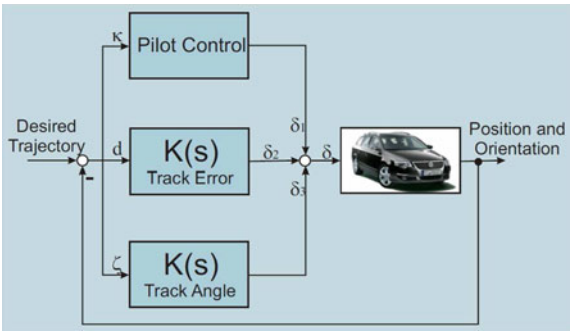


Fig. 24 Lateral control strategy

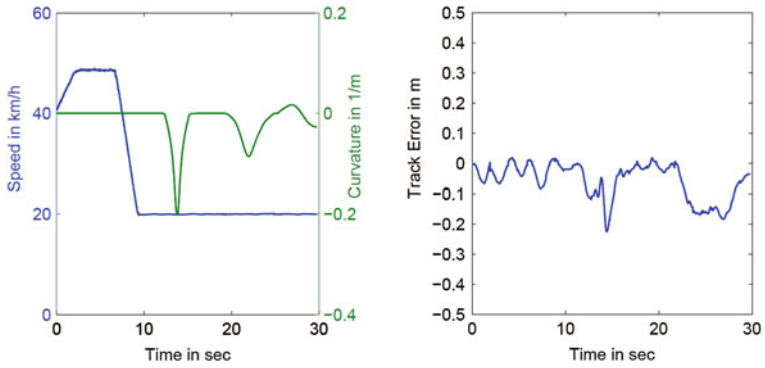


Fig. 25 Speed profile and track error of the trajectory

deviation (Fig. 24). In addition, a feed forward algorithm calculates steering angles based on the road's curvature that would be needed to follow the desired track based on parameters of the bicycle model.

Performance of the Lateral Controller: To demonstrate the performance of the lateral control strategy, a test track is used, where the vehicle starts on long straight trajectory and enters two sharp curves close to the turn radius of the vehicle. First, the vehicle is driving at a speed of almost $v = 50$ km/h. Both narrow curves are passed at a speed of around 20 km/h and corresponds to a lateral acceleration of up to $6\frac{m}{s^2}$ (Fig. 25).

During all tests and missions of the DARPA Urban Challenge, the control strategy was always stable with a low-track error, resulting in precise driving maneuvers.

6 Race Performance

Caroline managed to qualify for the Urban Challenge final event as one of the first vehicles, even though there were some problems during the NQE Mission A (merging into moving traffic) forced the team into some last minute software

changes. While missions B (urban driving, free navigation zones and parking) and C (urban driving, blocked roads and intersection behavior) were finished almost perfectly, the tight traffic setup provided by DARPA in mission A as well as the close proximity to existing k-rail barriers and some minor additional software bugs did call for some modifications.

First of all, similar to most teams, we had to significantly reduce the minimum safety distance to existing static obstacles in order to be able to follow the k-rail guided roadway which often called for a lateral distance to the barriers of less than 50 cm. Second, the planar sensor concept revealed some weaknesses when being confronted with condensed urban traffic. In that particular situation, three DARPA vehicles where approaching a T-intersection from the left side, while a single vehicle arrived from the right side with a little more distance. Both groups had been given precedence, our vehicle's objective was to pick a gap big enough to merge into the moving traffic. The vehicle stopped at the intersection and correctly waited for the group of vehicles approaching from the left side, but while they were passing the intersection, they physically blocked the field of view in the direction of the fourth target approaching the intersection from the right side. In the first attempts, our objective was to maximize safety and therefore we did not enter the intersection until the field of view in both directions was free enough. While this worked out well, it led to situations where the vehicle took sometimes considerably long to pass the intersection, since the provided DARPA vehicles where approaching in close proximity to each other. First attempts to make the vehicle a little more aggressive subsequently overshot the goal and lead to some unwanted precedence problems. The fifth attempt, however, lead to a setup compliant to our judges expectations and brought us into the Urban Challenge final event.

Within the final event, we successfully drove approximately 14 km within 2:30 hours (including pauses), successfully passing 65 intersections and 13 checkpoints. Three situations (some documented in news media coverage) are particularly interesting. In the first incident, the vehicle was leaving it's wanted trajectory by approximately 1 m on a steep-off road downhill section of the race track, calling for some human assistance in order to get it out of the sand berm and back on the road. At first we reasoned that probably a cloud of Mojave desert dust—which had conflicted with our laser-based sensors frequently before—must have led to a false detection which forced the vehicle off the road. Surprisingly, an analysis of our onboard camera material proved that the vehicle was approaching one of the chase vehicles waiting for some other robot having trouble while merging onto the two-laned road following the off road passage. Obviously, the vehicle had not been paused early enough on it's way down the hill in order not to conflict with the incident up in front and instead of hitting the chase vehicle, Caroline decided to try passing it closely on the left side—which ended as described above.

The second incident included the robot of the Massachusetts Institute of Technology in a left turn situation. Both robots were approaching the intersection at the same time with precedence for the MIT robot. Caroline did cut in

unexpectedly and forced both robots to be paused. After the race, the team found out that the precedence rules at that particular intersection were not modeled correctly by the team in the map material given prior to the race event, forcing Caroline to ignore the well-detected MIT robot and leading to the critical situation.

The third incident included the MIT vehicle again. Both robots where driving in a free navigation zone. Both robots were trying to find a way out of the zone when they approached each other very close. Neither robot carried out extensive evasive maneuvers as required by DARPA rules and therefore the bots had contact before being paused by DARPA officials. Since these maneuvers had been tested prior to the race extensively, the final explanation remains unclear to us. In the logfiles, a variety of smaller errors within the motion planning module could be found that may have led to unexpected behavior in the navigation zone. The collision decalibrated one of Caroline's front LIDAR units, and our robot was taken out of the race as one of the seven remaining robots in the field.

7 Summary

We presented the vehicle architecture, sensor and data fusion concept as well as the vehicle guidance system of the 2007 Urban Challenge vehicle (Caroline) of Team CarOLO (Technische Universität Braunschweig). The vehicle architecture is subdivided into four main functional levels for perception, situation interpretation, motion planning and motion control.

The perception approach is based on a hybrid fusion system, combining classical object-based tracking and grid-based data fusion for obstacle detection and roadway analysis, respectively. A variety of sensor types are applied including radar, fixed beam LIDAR, rotating laser scanners and camera systems. For the tracking system, the classical (e.g. rectangular-shaped) object model was replaced by a polygonal free-form approach suitable for arbitrary obstacle description. The target dynamics estimation algorithm was adopted to cope with the complex shape model. For the grid-based data fusion, the drivability of the roadway is described by a Dempster–Shafer probability mass model targeting to classify the environment into *drivable*, *undrivable* or *unknown*. LIDAR-based distance measurements and camera based colorspace segmentation results are transformed into and fused within the Dempster–Shafer drivability space. Data fusion allows higher robustness in order to provide safe lateral guidance of the vehicle through the roadway.

Following the perception layer, situation assessment and motion planning is carried out by an interrupt extended behavior-based approach. Depending on the current driving mode (e.g. free driving, parking or u-turns) an optimized trajectory is calculated and realized through the low-level control units.

The system described in this chapter has been profoundly tested and proved it's quality within the preparation and the final competition of the Urban Challenge event, leading to a ranking of Caroline within the top-ten vehicles.

References

1. Bachelder, I.A., Ullman, S.: Contour matching using local affine transformations. In: Proceedings of Image Understanding Workshop, pp. 798–801. Morgan Kaufmann, San Mateo (1992)
2. Bar-Shalom, Y.: Update with out-of-sequence measurements in tracking: Exact solution. IEEE Trans. Aerosp. Electron. Syst. **38**(3), 769–778 (2002)
3. Becker, J.C. Data fusion of object-recognizing sensors of an autonomous vehicle (German). Ph.D. Thesis, Institut für Regelungstechnik, TU Braunschweig (2002)
4. Berger, K.: Recognizing driveable terrain—an analysis of clustering methods. Studienarbeit, Institut fuer Computergrafik (2002)
5. Blackman, S.: Multiple-target tracking with radar applications. Artech House, Norwood (1986)
6. Bresenham, J.: Algorithm for computer control of a digital plotter. In: Seminal graphics: pioneering efforts that shaped the field, pp. 1–6. ACM, New York (1998)
7. Comaniciu, D., Meer, P.: Robust analysis of feature spaces: color image segmentation. In: Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 750–755 (1997)
8. DARPA: Urban challenge technical evaluation criteria. http://www.darpa.mil/grandchallenge/docs (2006)
9. DARPA: Urban challenge rules. http://www.darpa.mil/grandchallenge/docs (2007)
10. Dempster, A.: Upper and lower probabilities induced by multivalued mapping. Ann. Math. Stat. **38**, 325–339 (1967)
11. Dickmanns, D., Behringer, R., Dickmanns, T.: The seeing passenger car vamors-p. In: Proceedings of IEEE Intelligent Vehicles '94 Symposium, IEEE International, paris (1994)
12. Effertz, J.: An expandable multi-sensor data-fusion concept for autonomous driving in urban environments. J. Aerosp. Comput. Inf. Communi. **4**(12), 1108–1116 (2007)
13. Effertz, J.: Autonomous driving in urban environments through combination of object- and grid based perception models (German). Ph.D. Thesis, Institut für Regelungstechnik, TU Braunschweig (2009)
14. Elfes, A.: Occupancy grids: a stochastic spatial representation for active robot perception. Autonomous Mobile Robots, pp. 46–57. IEEE Computer Society Press, Los Alamitos (1991)
15. Foryth, D., Ponce, J.: Computer Vision: A Modern Approach. Prentice Hall, Upper Saddle River (2002)
16. Hoffman, J., Murphy, R.: Comparison of bayesian and Dempster–Shafer theory for sensing: a practitioner's approach. In: SPIE Proceedings on Neural and Stochastic Methods in Image and Signal Processing II (1993)
17. Klose, F.: Sensor fusion of areal environment data with occupancy grids (German). Master's Thesis, Institut für Regelungstechnik. TU Braunschweig (2008)
18. Maehlisch, M., Kauderer, T., Ritter, W., Dietmayer, K.: Feature-level video and multibeam lidar sensor fusion for full-speed acc state estimation. In: Proceedings of 4th International Workshop on Intelligent Transportation (WIT 2007), Germany, Hamburg (2007)
19. Mitschke, M., Wallentowitz, H.: Dynamik der Kraftfahrzeuge. Springer, Berlin (2004)
20. Rauskolb, F. et al.: Caroline: An autonomously driving vehicle for urban environments. J. Field Robotics **25**, 674–724 (2008)
21. Rosenblatt, J.: Damn: A distributed architecture for mobile navigation. Technical Report, Robotics Institute (1997)
22. Thrun, S., et al.: Stanley: The robot that won the darpa grand challenge. Springer tracts in advanced robotics—The 2005 DARPA Grand Challenge, pp. 1–43 (2005)
23. Thrun, S. et al.: Stanford's robotic vehicle junior: interim report. Stanford University, Techical Report (2007)

24. Urmson, C., et al.: A robust approach to high-speed navigation for unrehearsed desert terrain. Springer tracts in advanced robotics—The 2005 DARPA Grand Challenge 45–102 (2007)
25. Urmson, C., et al.: Tartan racing: a multi-modal approach to the darpa urban challenge. Urban Challenge Semifinalists Technical Paper (2007)
26. Zou, Y., Ho, Y., Chua, C., Zhou, X.: Multi-ultrasonic sensor fusion for mobile robots. In: Proceedings of the IEEE Intelligent Vehicles Symposium 2000, pp. 387–391 (2000)

Part V
Development and Test

Engineering Autonomous Driving Software

Christian Berger and Bernhard Rumpe

Abstract A larger number of people with heterogeneous knowledge and skills running a project together needs an adaptable, target, and skill-specific engineering process. This especially holds for a project to develop a highly innovative, autonomously driving vehicle to participate in the 2007 DARPA Urban Challenge. In this contribution, we present essential elements of a software and systems engineering process to develop a so-called artificial intelligence capable of driving autonomously in complex urban situations. The process itself includes agile concepts, like a test first approach, continuous integration of all software modules, and a reliable release and configuration management assisted by software tools in integrated development environments. However, one of the most important elements for an efficient and stringent development is the ability to efficiently test the behavior of the developed system in a flexible and modular system simulation for urban situations both interactively and unattendedly. We call this the simulate first approach.

1 Introduction and Motivation

Innovative research is often centered around interesting challenges and awards. The airplane industry started off with awards for the first flight over the British channel as well as the Atlantic Ocean. The human genome project, the RoboCups, and the series of DARPA Grand Challenges for autonomous vehicles serve this

C. Berger (✉) · B. Rumpe
Department of Software Engineering, RWTH Aachen University,
Ahornstraße 55, 52074 Aachen, Germany
e-mail: berger@se.rwth-aachen.de

B. Rumpe
e-mail: rumpe@se.rwth-aachen.de

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_10, © Springer-Verlag London Limited 2012

very same purpose to foster research and development in a particular direction. The 2007 DARPA Urban Challenge took place to boost development of unmanned vehicles for urban areas. Although there is an obvious direct usage for DARPA, there will also be a large number of spin-offs in technologies, tools, and engineering techniques, both for autonomous vehicles, but also for intelligent driver assistance systems.

Such a system needs to be able to understand the situation around the car, evaluate potential risks, and help the driver to behave correctly, safely, and, in case it is desired, also efficiently. These topics do not only affect ordinary cars, but also busses, trucks, convoys, taxis, special-purpose vehicles in factories, airports, mines, etc. It will take a while before we will have a mass market for cars that actively and safely protect the passenger and their surroundings from accidents in all situations, but their development is on its way.

Intelligent functions in cars are obviously complex systems. For a stringent deadline-oriented development of such a system it is necessary to rely on a clear, usable, and efficient development process that fits the project's needs. Furthermore, changing requirements and enhancements of technologies need to be incorporated into the development effectively. This kind of situation is well known in business and web-based software development. Therefore, that industry has developed appropriate methods and process frameworks to handle this kind of projects.

Among a number of agile development processes, extreme programming (XP), Scrum, and the crystal family of processes are the most prominent. However, these development processes are dedicated to software only and seem to not support traditional engineering processes properly at first sight, which often include the development and setup of embedded systems. Therefore, a compromising adaptation is necessary that addresses the needs of both worlds. Moreover, a coherent, efficient, and automatable tool suite is inevitable for a goal-oriented development project. We describe an efficient tooling infrastructure necessary for such a development project in Sect. 4. This tooling relies on a process for informal requirements to direct coding approaches which we describe in Sect. 2.

For quality management, quite a number of actions were taken. Among others, the steady integration, the requirement for continuously running code, and the regular integration into the vehicle are part of the plan. Most important, and this over time became a more and more obvious advantage in the "CarOLO" project which itself is described in Sect. 3, were the automated and unattended tests that had been developed for all parts and integration stages. Demanded by the test first development approach, tests for functions and algorithms were coded in a parallel manner to the actual code. The technology and measurements that enabled this testing process are described in greater detail in Sect. 4 as well.

The fulfillment of requirements for a safe autonomous car needs to be intensively tested. This is to a large extent achieved by running the software in a *virtual test-bed* for avoiding race conditions on valuable resources like the real vehicle. The system simulation acts like the real surroundings for the software by producing information such that it thinks it was on the street and decides accordingly.

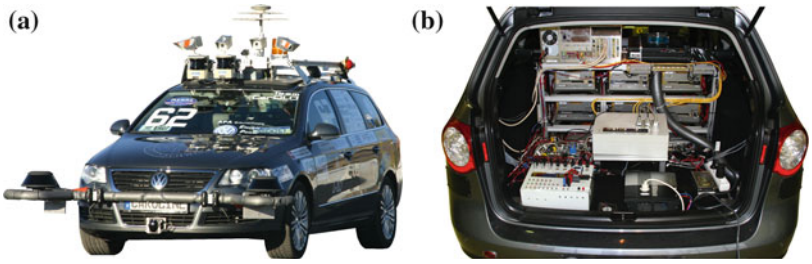


Fig. 1 A front and rear view of the vehicle "Caroline", an autonomously driving vehicle for urban environments (based on [13]): **a** for realizing an overlapping and thus a redundant field of view around the car to perceive the vehicle's surroundings, different sensors like single and multilayer laser scanners, radars, and cameras were used, **b** all data gathered from the sensors is transferred to Caroline's trunk, where different computers were used to process the data to derive driving decisions

This real world system simulation was developed not only to test the requirements, but also to allow the system to "virtually drive" through hundreds of different situations, which could contain other cars. The system simulation therefore allowed us to test—and especially to automatically re-test—the behavior of the software in a regression testing manner without any possible harm. We could not only test for positive situations, but also for negative and physically impossible situations of the surroundings and obstacles. Furthermore, the system simulation could be used in an interactive mode to understand a vehicle's behavior. This is one of the basic elements for actually being able to develop our autonomously driving vehicle as shown in Fig. 1, called "Caroline" in the "CarOLO" project, in time to participate in the 2007 DARPA Urban Challenge. The technical aspects of the system simulation used in the CarOLO project are described in [2] which formed the basis for this article. However, in Sect. 6, an improved successor is outlined.

2 Overview of an Efficient Software Engineering Process

Whenever things become complex, they need to be decomposed into smaller pieces. Also when the development of a product is complex, not only the product, but also the development activity needs to be structured to become manageable. Both in the areas of software development and systems engineering, a variety of such processes exists. An appropriate process can be chosen based on the needs of the project in terms of complexity, criticality, and urgency of the product. However, due to the different nature of "virtual" software versus physically existing hardware, these development processes differ greatly. Nowadays, this process gap is a constant source of problems. Before we look at the process used for the development of Caroline, we highlight a few of those distinctions.

Due to its immaterial nature, it is obvious that software can more easily be reorganized and evolved than physically existing hardware. Therefore, software

development processes can be iterative and incremental to allow an evolutionary evolvement towards a stable, robust, and mature result. Recent processes like XP, Scrum, or RUP advocate iterations in all stages with various durations. Iterations are necessary to continuously evaluate the current state of the customer's needs on the one hand. On the other hand, iterations enforce the continuous integration of existing artifacts into a *functional* system. Thus, iterations are a vital part to deal with the necessary improvement of existing software, changing requirements, and an incremental process that decomposes software into smaller pieces. The more innovative software products are, the less predictable a software development process is. In such a context many small iterations are preferable over a few larger ones. More and smaller iterations allow the software management to adjust priorities and to lead the project more successfully.

We believe that in integrated software and systems development projects with a heavy software component an iterative software development process needs to be used instead of the traditional engineering process. The best approach is to decouple subprojects as much as possible such that individual processes can be followed in the subprojects. A set of high-level milestones connect the subprojects and ensure a coherent process on the top, but within each subproject different forms of processes can be used. Furthermore, it is inevitable to decouple iterative software development from hardware, for example through virtualization. Looking into the software development process, we find that these many small iterations strongly imply a number of other development practices that are necessary to ensure progress and quality. However, most important is a continuous integration of the software pieces. Experience shows that software simply cannot be developed independently and integrated later, even if the interfaces are defined as precisely as possible, because patchwork inevitably must be done during integration that invalidates earlier development work. With modern tools for version control and configuration management, continuous integration can be achieved rather easily. The most tricky and hard to achieve part is to ensure that all team members are committed to a continuous integration process. This is because software must be developed in a collaborative fashion, where no code ownership exists and everybody releases and integrates his software at least several times a day.

Disciplined use of versioning is the basis for the next important process element, namely automated, integrated, and unattended testing. Testing is by far the most important technique for quality assurance and comes in many different flavors, beginning with unit tests to integration tests up to full system tests. Incremental software development requires periodic testing to test artifacts as well as their integration as already mentioned earlier. The "testing trap", which denotes unnoticed software bugs due to inconsequential software testing, can only be escaped through automated replaying of tests, which is also called regression testing, for each single increment. The important part of automated testing is not finding appropriate tests, but the techniques that run the current code to determine efficiently whether some property of the code is correct or wrong and without humans to run the test or interpret the results. As known from unit tests, automation helps

each developer to know whether an error was introduced to the code, in which iteration it was introduced, and in which part of the software. Error detection thus becomes much easier and after detecting an error, an identification of the location is relatively easy within a small area.

A very important aspect of automated testing is that it cannot only be used to test each iteration but even every single version. This means each development subtask, may it be a 10-min bugfix or a 3-h development of an algorithm, can easily be checked against all available tests at no additional cost of manpower. We therefore integrated the automated testing infrastructure with the version control system. Each version which was committed to the versioning server automatically triggered a testing process, where all available tests were run and feedback in the form of the number of failed tests with detailed information was given. Our experience is that in the long run this kind of quality assurance helps very much to foster efficient software development. Initially however, it needs a lot of discipline. Furthermore, appropriate tooling infrastructure is inevitable to make the developers accept this discipline.

Even more discipline is necessary to start with developing tests first for software elements as there should not be any untested functions in the system at any time. Fortunately, developers can later enjoy seeing automatic tests run through to status "green" at every new version. Testing also needs to deal with configuration variations. A common C/C++ problem allows code to behave differently on different platforms. The test first approach can help to identify problems early providing a so-called automated build farm for various platforms.

When developing autonomously driving cars, automated testing has a number of challenges to tackle. First of all software is integrated and closely linked with hardware, such as sensors and actuators, and through them to the surroundings. Appropriate abstractions for different parts of the software and the hardware are necessary to run tests efficiently. For the "intelligent part" of the software, it is not necessary to run tests based on full sensory input, but to provide distilled, aggregated information about possible obstacles as well as the pathway to drive through them. A highly important abstraction for efficient test automation is to replace the real hardware by a simulation. A simulation of the hardware allows automated tests on ordinary computers and is thus available for each developer independently. As all physical elements are simulated, it furthermore allows decoupling the time for running a software test from the real time. This allows running a complete city traversal in a few seconds. We are thus able to run thousands of tests for every new version each night. As a prerequisite we needed to develop a testing infrastructure that:

- Allows us to define various constellations and subsystem configurations of the software parts to be tested,
- Provides a software test-bed to probe the software under test and to understand whether it behaved correctly,
- Is capable of providing physics-based behavior of the controlled vehicle as well as the urban surroundings correctly and in sufficient detail, and

- Allows us to easily define new tests including automated evaluation of the test results.

Of course, tests that deal only with the simulation software are not enough to ensure a robust automotive system. Therefore, a continuous deployment of the software subsystem into the car and a (re-) run of a sufficiently large set of tests in the vehicle are inevitable. For that purpose, we had an additional test team that ran the fully equipped car in various traffic situations. The test team got software releases in regular iterations and checked the car's abilities against the requirements from traffic rules, DARPA urban challenge rules, and story cards mapping those requirements into concrete driving missions which is outlined in the following. However, a stringent software testing process considerably reduces the amount of time necessary to fully run the hardware-in-the-loop (HIL) tests. Furthermore, the complexity of traffic situations necessary to do the test usually requires several people in each test run. For example, any single test that deals with a correct behavior in a junction takes at least 15 min to setup, run, and check the resulting behavior. It involves several other cars and people to produce appropriate junction situations. Multiplied by the variety of junctions and the many possibilities of cars coming in different directions, this would take far too long to actually run all of them in reality. So quite a number of junction situations are tested only in the system simulation. The possibility to rerun those situations efficiently and automatically is important to ensure the stringent and effective development process needed.

Another important issue to be taken care of from the beginning is to organize the software in appropriate subsystems and components, to define the technical interfaces, and to take additional actions so that the software can be developed and tested independently. Only if the software obeys a number of best practices and design patterns, it is possible to test the software efficiently. For example we can decouple the time a test takes from the time the tested software thinks it runs in, if the software does not directly call the operating system about the current time or even worse, counts itself, but uses an adapter interface. Similar techniques are necessary, if outside software needs to be incorporated that does not have a testable software architecture, neighboring systems are not part of the testable subsystem, or sensors and actuators come into play. Architectural testing patterns help to develop software that can be tested in various configurations and parts individually, example mock objects or abstract factories for object construction.

Moreover, we have adopted some more practices, for example from XP, like short iterations. For example, one successful organizational and structuring tool were story cards. A story card describes briefly and explicitly the goals for a development iteration and thus leads to a useful, efficient, and focused structuring of the requirements and also the development project. Accompanied with a definition of measurable goals for every task, these story cards allow the developers to understand and measure development progress.

Having highlighted a number of special issues that arise when integrating an agile software and a classic engineering process, we note that classic engineering

and software engineering indeed have different development cultures. It takes a while until both cultures efficiently work together, but when properly integrated, the resulting output is tremendous and of very good quality.

3 The CarOLO Project

The CarOLO project was carried out from May 2006 until December 2007. In this project, the autonomously driving vehicle named "Caroline" was developed which participated in the 2007 DARPA Urban Challenge. It was among only ten other vehicles from more than 85 initial vehicles from all over the world which qualified for the Urban Challenge final event.

3.1 The 2007 DARPA Urban Challenge

The 2007 DARPA Urban Challenge was the third major competition and the successor of the 2004 and 2005 DARPA Grand Challenges [6, 22]. Compared to those earlier series in which the competitors had to develop a vehicle which was capable of driving autonomously through an a-priori unknown and rough terrain using a given digital map, the Urban Challenge increased the difficulty significantly.

In the Urban Challenge, not only did a digital map had to be processed to reach a given set of destinations, but there was an added requirement that the vehicle had to deal with dynamic vehicles. These vehicles were controlled both by human safety drivers and by other "robot" competitors as well. Moreover, the car had to obey logical traffic rules which define priorities at intersections or set speed limits for example. Furthermore, the car had to park without colliding with other already parked vehicles in a parking lot.

The semifinal and the final itself took place at the former George Air Force Base in Victorville, CA. For the challenge, DARPA had rebuilt that area to simulate an urban environment with junctions, intersections, and parking lots.

To participate in the challenge, DARPA provided two possibilities: the first was called "Track A" and was funded by DARPA, and the second called "Track B" was not funded at all by DARPA and was open to nearly anyone who could provide an American team leader for the project. The CarOLO project participated in the latter track.

Both tracks consisted of a multi stage participation process. The first major stage was to provide a video demonstrating the vehicle's capabilities at an early stage. In that video, the vehicle had to follow a road using the available lane markings and to stop right before an obstacle blocking its lane. After a short while, the vehicle had to start a passing maneuver and merge back into its own lane to finish the course.

Using the video, DARPA selected teams for a closer inspection at "Site Visits" which were at the performer's location. Due to the fact that the CarOLO project was initiated in Germany, the team and the vehicle had to be shipped to the USA for demonstrating its capabilities in June 2007. Therefore, our team cooperated with the Southwest Research Institute (SwRI) in San Antonio, TX.

The site visit demonstration started with an inspection of the emergency control system. Next, the vehicle had to follow its lane, pass a stationary vehicle as already shown in the video, show its U-turn capabilities, and behave correctly at intersections. Using this multi-stage participation process, DARPA could enforce a high quality for the semifinal.

For the semifinal, which took place in Victorville, CA as mentioned before, only 35 teams qualified after the site visits. The semifinal, which was called the "National Qualification Event", provided three independent test areas which tested different aspects of the vehicles.

The first test area, "Test Area A", provided an eight shaped course to test correct merging at T-junctions. Therefore, the vehicle had to find a gap in the moving traffic to merge into. The second test area, "Test Area B", tested a vehicle's capabilities to safely pass any obstacle at the side of a road and to park the car in a parking lot. The last area, "Test Area C", re-tested correct behaviors at intersections and dynamic re-planning in the case of blocked roads.

Based on the vehicles' performance in the semifinal, only 11 teams qualified for the 2007 DARPA Urban Challenge final event which took place on November, 3 [16].

3.2 Team CarOLO's Caroline

The vehicle shown in Fig. 1 was named "Caroline". It is a 2006 Volkswagen Passat station wagon which provided the necessary technical capabilities, like drive by wire, to realize our software and system architecture. For a technical explanation with greater details we refer the reader to [1] and [13].

In Fig. 2, the layered software architecture of Caroline is depicted. The architecture consists of several loosely coupled modules for solving different tasks like detecting lanes, fusing incoming new sensor data, or updating the currently planned route. The entire system used the so-called *pipes-and-filters* pattern where one module processes incoming data for producing outgoing data which is the new incoming data for the following module. Thus, the task of driving autonomously was split into smaller pieces obeying the "separation-of-concerns" rule supported by a self-developed software framework.

The software framework provided rudimentary system services like concurrency and communication. Using this framework and the layered pipes-and-filters architecture, all modules could simply be reused for system simulation. Technical details for the system simulation used in the CarOLO project can be found in [2]. In Sect. 6, a new approach is presented for efficiently designing, using, and

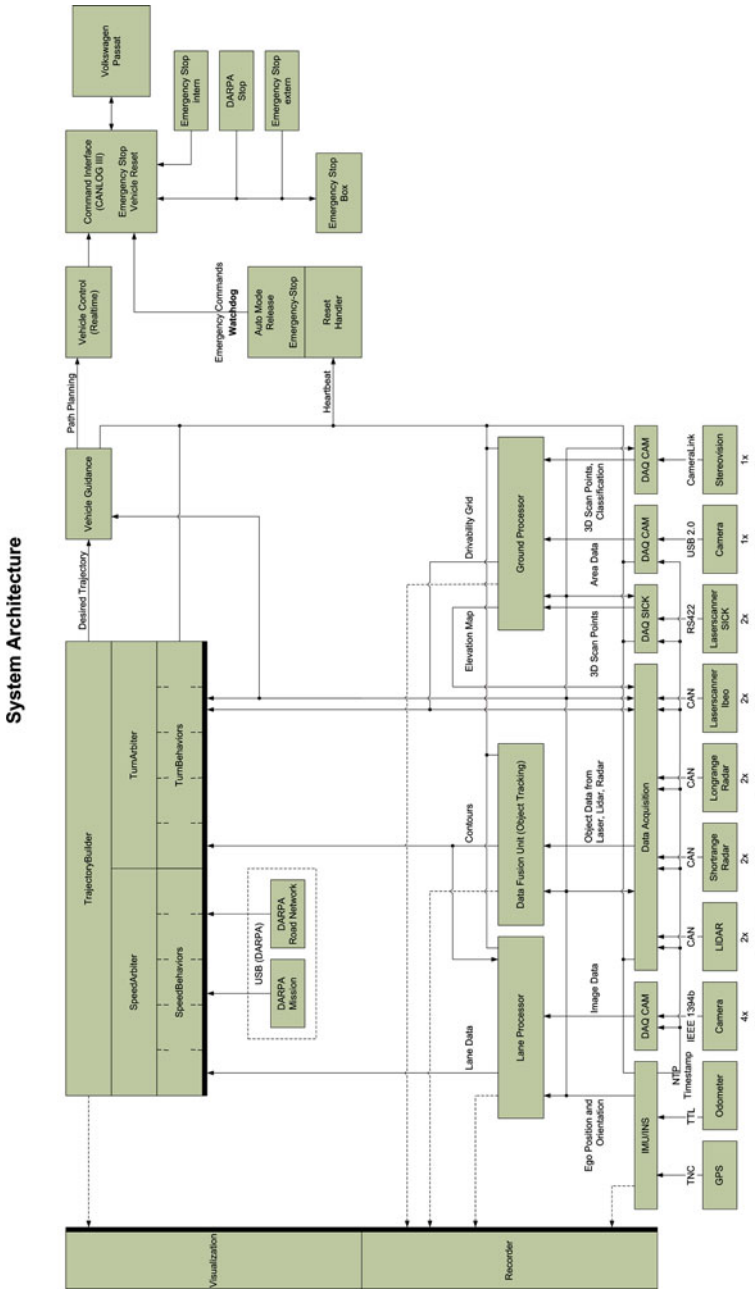


Fig. 2 System architecture for the autonomously driving vehicle "Caroline" (based on [13])

unattendedly executing system simulations which avoids some conceptual design problems from the software framework used in Caroline.

4 Tools for an Efficient Software Engineering Process

As is the case with the development of any complex software-intensive systems, the software development process described in Sect. 2 can only be accomplished if an appropriate tooling infrastructure is available. This is important because to participate in the Urban Challenge, the complete software and hardware system had to be developed on a very tight schedule because there were no negotiations at all. For a successful participation, both efficiency and quality of actions had to be balanced wisely. These special circumstances led to the organizational implementation of the agile software and system engineering process based on XP. Clearly, a modern IDE like Eclipse is used for direct code development in C++ and MATLAB/Simulink for the control algorithms. However, as MATLAB/Simulink does not scale for complex data structures or supports properly the design and implementation of software architectures, most of the code by far was written in C++. Therefore, in the following, we concentrate on the C++-part of the software development.

For planning purposes, plain mathematical functions were used to describe the algorithms, and UML class and deployment diagrams as well as statecharts were used for the software architecture as well as the important state-based behaviors and data structures. However, these diagrams were used for discussion and therefore served as documentation only.

As described, milestones were centered on story cards, as shown in Fig. 3, that served as a definition of measurable goals. These measurable goals were the base for a consistent test process and its tooling infrastructure that is described in the following. The whole team was distributed in several locations and team members sometimes worked in different time zones. For a uniform understanding of progress, a single source of information was necessary for collecting tasks, tracking bugs, and publishing progress. Using Trac as an integrated and easy to use web based portal for the complete software and system development process enabled the team to track changes to the software over time, and evaluate the actual state of the software generated by the back-end tool chain. As mentioned above, every story card was virtually available for every team member at any time to see the most important aspects for the current and next development iteration. In addition to the description of the next iteration, a list of tasks each with a metric of its completion and a list of open bugs where available for every virtual story card.

For achieving the goals described in a story card, among other things, system simulations as described in Sect. 6 could be used by developers to test their code. In an initial step, the requirements of the story card were translated into an initial number of tests which described a *scenario* even before the software was developed. After fulfilling the requirements in the system simulation, the software was put into operation on the real hardware.

CarOLO –StoryCard MS10-SC01

Beschreibung:

Meilenstein 10

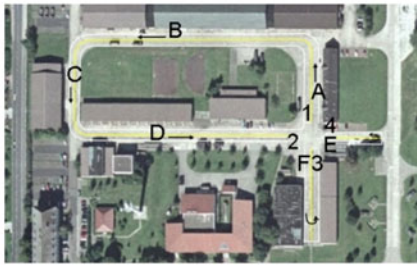
Termin 26. April 2007

Titel **Warteschlange**

Ort Site Visit-Kurs, Campus Süd, Braunschweig

Beschreibung Start bei A, Folgefahrt bei C, Queueing bis zur Kreuzung. Fremdfahrzeug passiert nach E, Caroline stoppt.
Kreuzung frei, Caroline passiert nach A, fährt eine Runde. Bei 1 wartet bereits F-Fzg. Caroline stoppt, lässt F-Fzg. nach 3 passieren und biegt selbst nach A ab. Weitere Permutationen durchspielen!

A.9, B.2, B.4



Modul	#	MS	Beschreibung	Verantwortlich	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do
					16.04	17.04	18.04	19.04	20.04	21.04	22.04	23.04	24.04	25.04	26.04
KI	KI-1	10	Erkennung der Kreuzungsreihenfolge	homeier											
	KI-2	10	Queueing	homeier											
	KI-3	10	Überholt nicht bei Gegenverkehr	homeier											

Fig. 3 An example of a story card which describes the requirements in an intuitive manner for one scenario. This card is inspired by Volère's requirements' specification. On the first line, a unique identifier for this scenario is provided. On the *left-hand side*, some information like a headline, date, and location are defined. This information is followed by a detailed description of the behavior which is expected from the vehicle to fulfill this scenario. On the *right-hand side*, an image illustrating the scenario and its situations is depicted (for our example an aerial image of our testing site which is located on the former barracks of the German Forces is shown). In the bottom area, the natural description is split into detailed and identifiable *functional* requirements for every module. Further columns are used for responsibilities and daily fulfillment rates; the actual milestone is indicated by the last *green* column

To manage parallel development as well as different configurations in the system simulation and real hardware, a release and configuration management tool based on Subversion and Fast System Versioning (FSVS) [21] was used. This allowed us to progressively enhance the software development in small increments, while at the same time restore older, stable versions for demonstrations and events, where a stable and demonstrable software version needed to be loaded on the car. Furthermore, the configuration management had to ensure that hardware changes fit the loaded software releases. As usual, all seven car computers were not only backed up in their own configuration, but also version-controlled.

Using FSVS as a version control system for filesystems enabled the team to simply and safely test new software versions and to maintain the integration between parallel developments as well as track open issues and potential bugs which were found during real vehicle tests. With version control the independent test team had the ability to retrieve specific software releases that the development team wanted to be tested. This further decoupled testing and development and allowed more parallelization and thus increased efficiency. All car computers were consistently restored to the specific software release and a detailed test process

based on the measurable goals of the virtual story cards could be carried out rather efficiently. In particular, bugs and behavioral issues could be recorded in such a way that they could be replayed and analyzed in detail if necessary. Both the development and the test teams could simply restore the development state of the car in the desired software release by switching every car computer to the appropriate revision using one simple command. Instead of FSVS, more recent filesystems like ZFS or Btrfs could be used for a similar purpose.

The combination of virtual story cards and consistent release and configuration management enabled the team to safely develop and test potentially dangerous new software functions without breaking an already running software system on the vehicle. Furthermore, the list of open or closed tasks allowed the project management to get a current impression of the project's status. Appropriate tools which were used in the project for developing Caroline were:

- Specifying story cards: Microsoft PowerPoint.
- Modeling algorithms: UML tools.
- Realization: Eclipse for C++ (CDT), MATLAB/Simulink.
- Version and configuration management: Subversion.
- Unit testing: CxxTest [20].
- Requirements testing: system simulation using scenario-based executable test drive specifications.
- Deployment: Unix-based scripts developed within the project combined with FSVS.
- Software build and integration process: Make/QMake/Cook.
- Milestone planning and tracking: Trac [8].
- Bug tracking: Trac.
- Knowledge management: Wiki provided by Trac.

Our software and systems engineering process relied on a variety of software development tools and some customizations and extensions to combine these tools and to optimize our tool chain. As mentioned before, elements of XP, like the test first approach, common code ownership, pair programming, and continuous integration were the basis for an efficient and successful development process. Reducing integration activities to nearly zero through the continuous integration principle was one key element in our development process. As described earlier, this implies that every developer had to integrate his work frequently and was disciplined in using the controlled source code repository based on the version control system.

Hence, Subversion managed nearly everything necessary to build the project to ensure self-containedness of the entire project. From install scripts, property files and test scripts, up to IDE configurations, the repository contained and provided all project-dependent data. This enabled the team to fully build the system from just a checkout on a machine with only a minimum amount of pre-installed software like some third party libraries. Moreover the use of Subversion and the software build tools allowed us to setup the fully automated build process, including test runs, which acted as a monitor to the repository needed for quality assurance.

With this approach we were able to find errors quickly and fix them as soon as possible. Triggered by every commit against the repository, central servers started and checked out the project sources and initiate an entire build. This is the point where software construction tools came into play. On the developer site these tools helped to speed up the build process and at the same time ensure a consistently built result by analyzing changes and dependencies to identify what really needed to be rebuilt. On the server side it allowed us to build alternative targets for different forms of use, so we were able to run a system build with or without test code. Furthermore, the testing system separated time consuming high level tests by detaching the complete automated test run to be done in a parallel manner on different servers. Thus, whenever a developer checked in a new version of the software the complete automated set of tests was run.

Feedback of the automated and continuous build and test process was sent to the developer through email and through publication of the build and test results on the project specific portal web site using Trac. This greatly improved responsiveness of the test system. Problems did not remain undetected for a long period, and in only a short time faults were bounded to a small portion of changed code. Efficiency of the development process as well as a responsible and self-disciplined form of development were effectively assisted by the tooling and testing infrastructure.

Figure 4 shows the workflow of our multi-level build process. The fully automated process for checking the software quality consisted of several consecutive steps. Starting with a compile check, compilation as well as syntactical conflicts were detected pretty early. However, it was expected that code which could not be compiled was never checked into the version control system.

To automate tests, we used a light-weight and portable testing framework for C++, called CxxTest. During the test run, the memory checker Valgrind searched for existing and potential memory leaks in the source code. An additional tool from the GNU compiler collection called GCov was used to report the test coverage of the source code. While running the test code it counted and recorded executed statements. The intent was to implement test cases which completely covered identified critical parts of the software.

Tests were usually implemented directly in C++. Experience has shown that for a disciplined test definition in a project, it is very helpful that the implementation of tests is done in the same language. This enables an integrated test development process and avoids the hurdle of learning another testing notation. The failure of any single test case caused the complete build to fail and immediate feedback was given to the developers. In order to check real-time properties, which are of course necessary for a timed execution in a traffic situation, a step for profiling was done to check the computation time.

For an initial phase, we found it helpful to add another step to the build process, which checked some compliance rules for the coding guidelines. For example, it analyzed the source code for appropriate definitions of names of variables and classes, checked depth of inheritance, number of attributes, sizes of method bodies, appropriateness of indentation, existence of comments, and the like. But moreover,

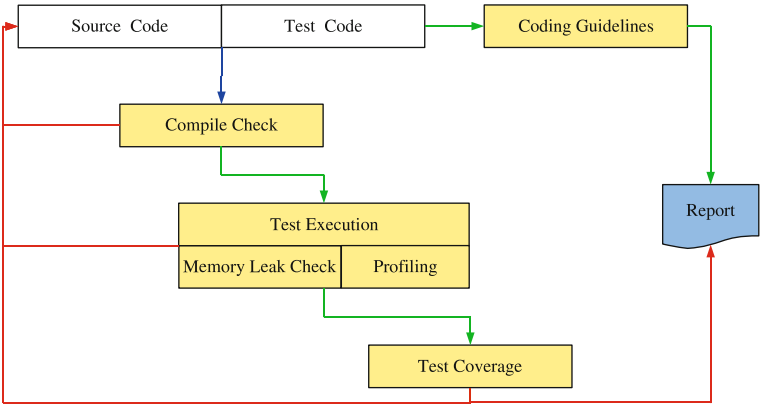


Fig. 4 Multi-level test process: the first stage contains the compilation check, followed by a memory leak check which is executed during the test run to find potential memory leaks or NULL pointer access. After executing the test cases, their coverage of the source code is computed. Here, different coverage criteria, like statement coverage or path/branch coverage, can be defined and checked. These coverage criteria indicate where the developer should add or modify existing test cases to fulfill the required coverage level. After fixing bugs, memory leaks, or adding more test cases, the algorithm can be optimized using profiling tools. Finally, the code can be checked using a given coding guidelines definition. All results are aggregated into one single report. This multi-level test process can be stepwisely executed manually by the developer or completely unattendedly and automatically using the continuous integration system

even complicated guidelines like visibility rules through inheritance could also be checked. This step could be automated as well [4].

The process outlined above not only worked automatically at every check-in, but also could be executed manually by every developer. When starting a commit cycle the developer first updated his working copy, ran all tests, and then committed his changes only if everything built well and all tests ran without errors. This lead to a double check, both before the developers committed and automatically at the server side by the master build process. This was reasonable because there was always a chance that the repository was not properly updated by the developers.

As a result of this approach, developers could easily rerun tests and detect many bugs or inconsistent enhancements locally and rather quickly. Thus, fixing discovered bugs was done rapidly and we had a stable and properly working system almost all the time. The benefit was that everyone shared a stable base to start development. Frequent commits, usually more than once a day, guaranteed that re-integration of newly developed or enhanced code did not take long and we usually had little integration overhead.

5 Application of the Software Engineering Process in the CarOLO Project

As outlined in [2], we used the system simulation not only for interactive development of an artificial intelligence module in the CarOLO project but also as part of an automated tool chain running on our servers over night. The main goal behind this application was to ensure a certain level of software quality. Therefore, we extended the test first approach mentioned in Sect. 4 by using the system simulation shown in Fig. 5. We called this approach the "simulate first approach".

Almost any development process starts with analyzing the requirements documents provided by the customer. In the CarOLO project, we used the DARPA Urban Challenge documents to understand the requirements. These documents contained mostly abstract and non-functional definitions for the autonomously driving vehicles. On the one hand, these requirements were rather stable—even though they were occasionally changed. On the other hand, they were rather vague and left open a lot of options for possible traffic situations, weather conditions, forms of roads, etc. Three big phases led the project from an initial setting through the first application (end of phase 1), through the site visit (end of phase 2) to the National Qualification Event (NQE) and Final Event (phase 3) as already described in Sect. 3. Actually, in any good development project there is a final phase with a retrospective and a discussion of the insights, the gathered knowledge and experience, and a consolidation of the developed software. This also includes the identification of reusable parts of the software. Each of the first three phases is broken up into several small iterations.

In every iteration a new task dealing with a coherent requirements group was chosen by the development team, prioritized in the product backlog, and defined using the Scrum process for agile software engineering as mentioned in Sect. 2. These requirements were refined into the already discussed story cards and scenarios which were designed for both a virtual test drive and a real vehicle test for evaluations using the completely equipped car. This early definition of the virtual test drive forced developers to clarify general parameters and conditions before starting their implementation. The result was an *executable test drive specification* that tested all requirements to be implemented. With this, the implementation of the system and the virtual tests could run in a parallel manner.

In the testing phase, after designing a virtual test drive, the availability of necessary validators was checked. If there was a condition which was not handled by a required metric, an appropriate validator was implemented. As mentioned earlier, these validators were the base for automated test runs, which were necessary for the continuous integration of the complete software system and therefore a vital part of a consistent software system engineering process.

The newly implemented test cases were grouped together in a test suite and formed an *executable specification* of the virtual test drive. The new test suite was then integrated in the tool chain. None of the old test suites should have failed and only the new ones should not have passed. With the failed tests, the

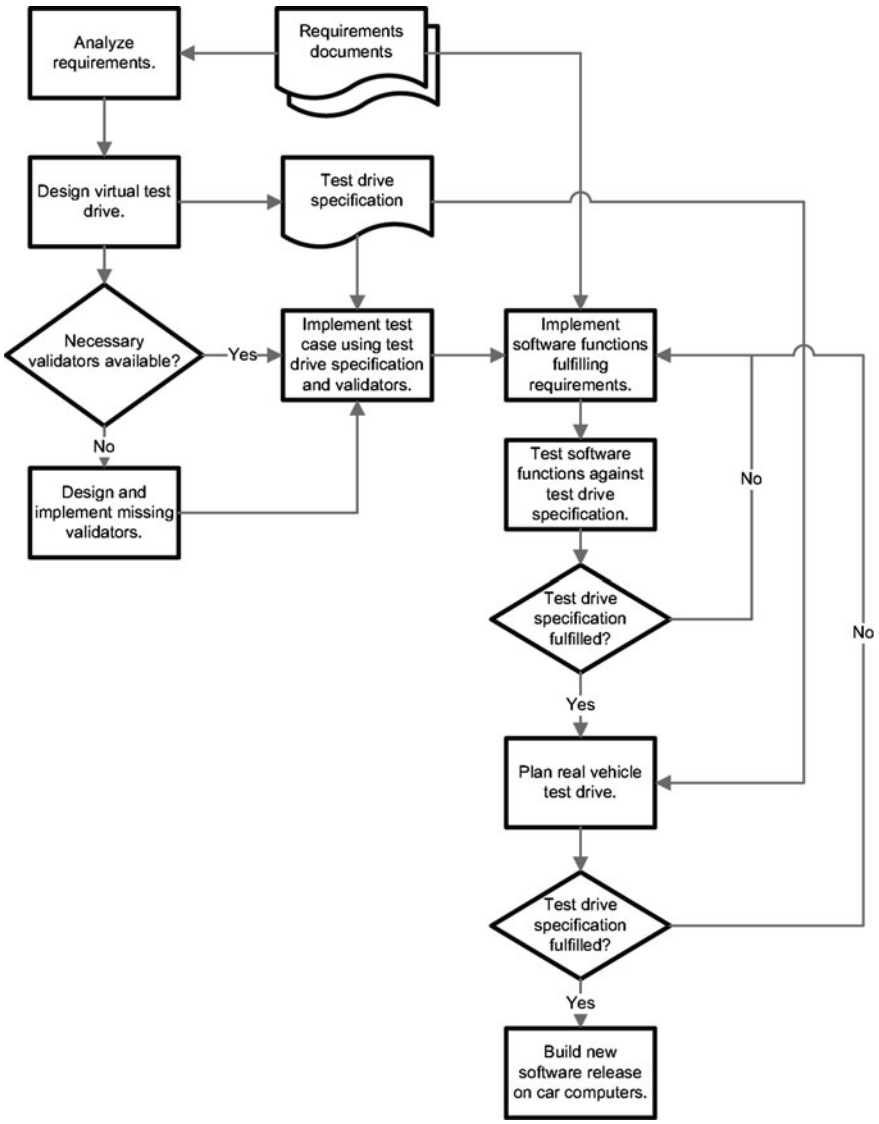


Fig. 5 Simulate first approach inspired by the unit testing approach: here, the customer's requirements are part of the technical software development process because they are the basis for defining scenarios to be used in interactive simulations. For setting up unattended simulations, not only the customer's needs are necessary, but even more, his acceptance criteria have to be discussed to define so called validators which continuously supervise a running system in a system simulation to evaluate whether it fulfills the customer's needs or not

implementation of the new artificial software functions began. In small iterative development steps the software module was extended for fulfilling every test case of the new test suite.

Although the above sounds like a normal test first approach, there are a number of differences. First of all, the test suite which captured the high level requirements for handling traffic usually did not change the signature of the overall system. The reasoner as well as the system simulation had stable interfaces and only needed to change its behavioral algorithms. So we did not need to define interfaces before the tests were defined. And second, those high level tests were black-box and did not rely on the internal structure of the code. However, for a thorough test of the reasoner, it was helpful to add more system simulation based tests after the code was implemented to check the states and transitions between them that occurred in the reasoner as well as traffic situations that the reasoner had to handle.

As with usual test first approaches, these small steps were iterated until the complete test suite was satisfied. After completing the implementation, the new intelligent software functions fulfilled the requirements in the virtual test drive. Subsequently, the test drive was planned and executed using the real vehicle when applicable.

If the test drive was successful the new software could be released and marked as stable. After the test drive, usually optimizations needed to be implemented, bugs fixed, and technical issues or misunderstandings from the requirements documents fixed. Before the code was modified, the system simulation was used again to extend the test suite in such a way that errors became visible under a virtual test and then the software was modified. These virtual test drives could be repeated at nearly no cost and helped the team to quickly and on time develop the necessary software functions.

In a second variant, we enhanced our system simulation environment so that multiple car instances could be driven in the same world model. Using multiple instances allowed us to run several intelligent cars together. On the one hand this was a good way to investigate artificial learning of optimal driving behavior in a potentially dangerous world. On the other hand, we could handle race conditions by running multiple instances of the artificial intelligence that started from the same position in the world data with the same mission, i.e. drive the same route. Starting multiple instances with the same mission data allowed us to understand and compare the performance of several versions of our intelligent car. It was also possible to measure the stability of the intelligent algorithms over time when using the same software revision in slightly enhanced courses. Furthermore, it was possible to watch the virtual Caroline becoming an even more optimal driver based on the increasingly optimized revisions of the artificial intelligence modules.

6 The Hesperia Framework: An Integrated Approach for a Virtualized Software Engineering

In the CarOLO project, we developed and used a specialized software framework for system simulations as outlined in [2] and [13]. However, this framework had some design drawbacks. One important issue was the chosen communication concept. In that software framework we chose TCP for reliability purposes to send data from one component to another. TCP allows a stream-oriented, directed, and most importantly, synchronized communications. These characteristics are suitable for different contexts; however, in the context of autonomous driving, where several components are sending compact messages with a high frequency, another protocol is more suitable.

Moreover, TCP does not allow broadcasting messages only once for several receivers. Hence, the same message is sent twice or even more often depending on the number of receivers and thus, the data is redundant on the network which might cause packet collisions, consume valuable bandwidth, and even worse cost transmission time.

Furthermore, the communication based on TCP is directed. Thus, all modules know their communication counterparts. Therefore, the system architecture must be defined a priori in a very mature manner to avoid frequent changes to communication paths between all modules. And even more, if another component *C* needs the same data from a component *A* which is already sent to a component *B*, component *A* must be modified to fulfill component *C*'s needs.

Moreover, the software framework used in the CarOLO project did not provide a formal specification to define scenarios with different situations in a machine processable manner. Thus, the setup of virtual test scenarios was not supported well.

Therefore, these conclusions and results yielded the development of an integrated approach for virtualized software engineering [3]. This approach is called "Hesperia"[1] realizing a multilayer software framework which is outlined briefly in the following.

6.1 System Architecture of an Autonomous System

In Fig. 6, the general system architecture of an autonomous system is shown. Such a system, which is embedded in the system's context, consists of three main layers: the perception layer, decision layer, and action layer.

The gray inner box depicts the complex embedded system itself consisting of the aforementioned three processing layers. The system itself can be supervised

[1] The name "Hesperia" is derived from a town in CA where team CarOLO was accommodated.

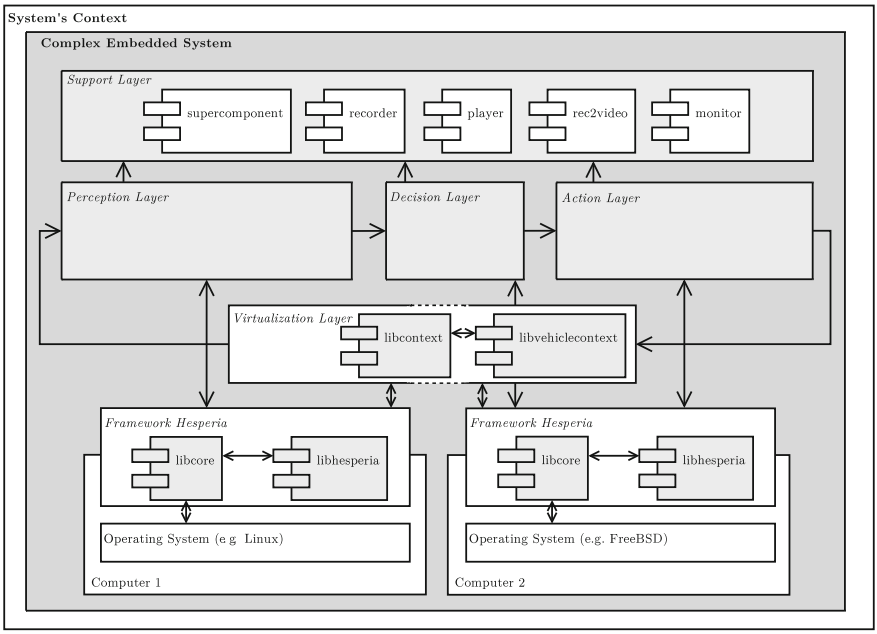


Fig. 6 System architecture for a complex embedded system with sensors and actuators. In the center the main processing components are depicted: perception, decision, and action. The first layer perceives the system's surroundings for extracting relevant features or abstracting from the incoming raw data. The second layer analyzes and interprets the preprocessed data for deriving a set of actions which are processed by the last layer by using control algorithms, for example. To support inspecting the system non-invasively, the support layer can monitor a currently running system or record and replay previously captured data. For setting up interactive or unattended system simulations, the previously described processing chain must be closed. This is realized by the virtualization layer provided by the framework Hesperia

using tools from the support layer. Some important tools are *recorder* for recording *non-reactively* data exchanged by the running components and *player* for replaying the data. Moreover, *monitor* is available to visualize complex data flows like the perceived sensor raw data or a computed trajectory to be driven by the vehicle. This component also does not interfere at all with the three aforementioned layers compared to the software framework developed in the CarOLO project.

To allow interactive as well as unattended system simulations, a layer for *closing the data processing loop* is necessary. This layer is called the *virtualization layer* due to its main task to simulate either missing components of the real system, such as sensors, or to modify a system's context, such as other moving vehicles ,using a mathematical model.

Technically, any component can be virtualized without modification using the library *libcontext* which is responsible for controlling the overall system time and the entire communications. Thus, the running system under test is decoupled from the real system time and can be run faster or slower if necessary. Thus this library,

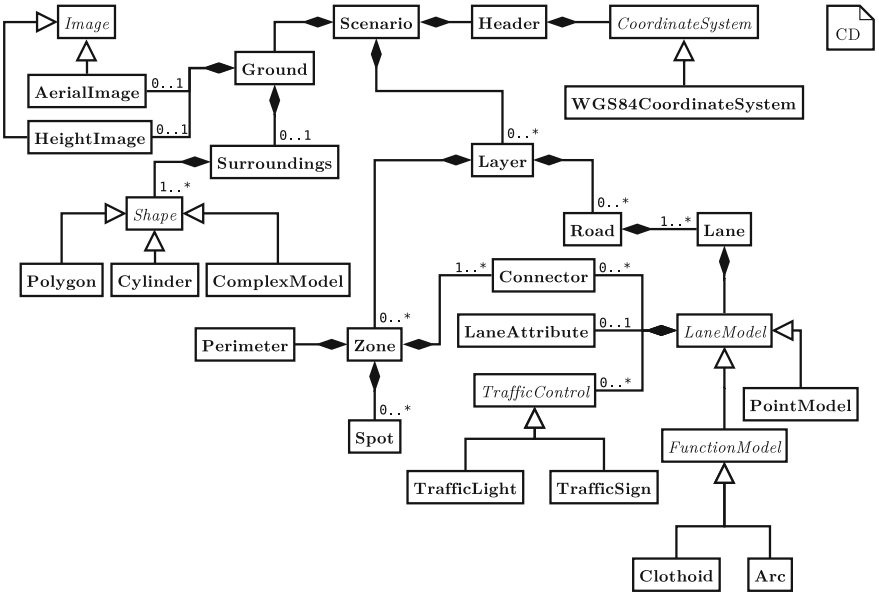


Fig. 7 Excerpt from the formal description of a scenario. This UML diagram is the base for a domain specific language

as part of the Hesperia software framework, is the enabling technology for re-using unmodified components in unattended and repeatable system simulations.

For providing suitable mathematical models of the driving behavior of vehicles, the library *libvehiclecontext* was developed. This library can be used for interactive as well as unattended system simulations.

6.2 Modeling the System's Context

For defining repeatable and meaningful system simulations, a formal description of scenarios and situations is necessary. This input data can be derived, for example, from the aforementioned story cards. In Fig. 7, an excerpt of the formal description of a scenario is depicted.

This formal description consists of more than 25 entities for describing the stationary system context and nearly 20 entities to define dynamic elements like other vehicles with their associated behavior. To use this formal description in a machine processable manner, a domain specific language (DSL) was derived. The DSL was designed using experiences from the 2007 DARPA Urban Challenge as well as conditions from a context of a vehicle which is driving automatically on highways [23].

For optimizing the DSL itself, its first implementation was made using the MontiCore framework for developing and using DSLs [10]. Thus, the language

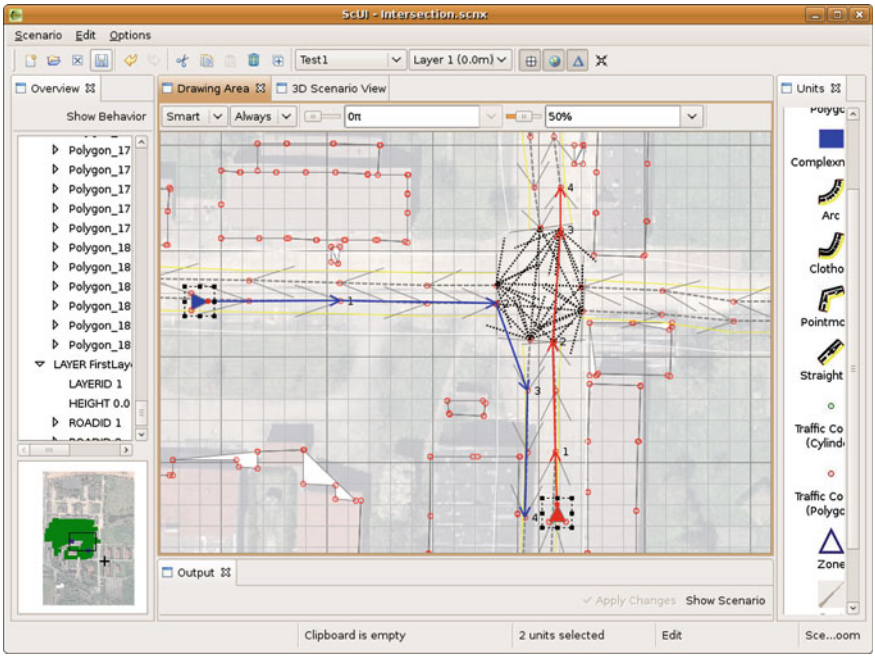


Fig. 8 Graphical editor for using the DSL realized with the Eclipse rich client framework. In the lower left corner, a small bird's eye view map of the scenario is shown. On the left hand side, a hierarchical representation of the DSL's instance of the currently edited scenario is shown which can be modified by the user. In the center of this figure, the main editor view is depicted which shows a situation at an intersection modeled for two vehicles. On the right hand side, a set of elements is available which can be added to the scenario by simply using drag and drop

could be simply enhanced and modified to be suitable for use in automatable system simulations. Furthermore, a graphical editor, as shown in Fig. 8, based on the Eclipse rich client framework [19], was realized easily by re-using the Java sources generated by the MontiCore framework [17]. In that figure, a traffic situation at an intersection is shown.

For processing the DSL in the Hesperia software framework, a C++ lexer and parser were necessary. To avoid additional tooling in the software build process, which may be error-prone or cause inconsistencies between the generated classes from the DSL and the classes using these generated ones, a compile-time generated implementation was chosen based on Boost spirit [5]. Here, the definition of the DSL is provided by a C++ template specification which is read by the C++ compiler itself to generate the necessary classes for the non-terminals and terminals of the language. This Boost spirit framework itself is wrapped in Hesperia to encapsulate the access to the classes of the abstract syntax graph (ASG) as well as to handle errors in the given DSL's instance appropriately.

The DSL is a core part of Hesperia for defining interactive or unattended system simulations. Therefore, not only abstract elements like lane markings or traffic

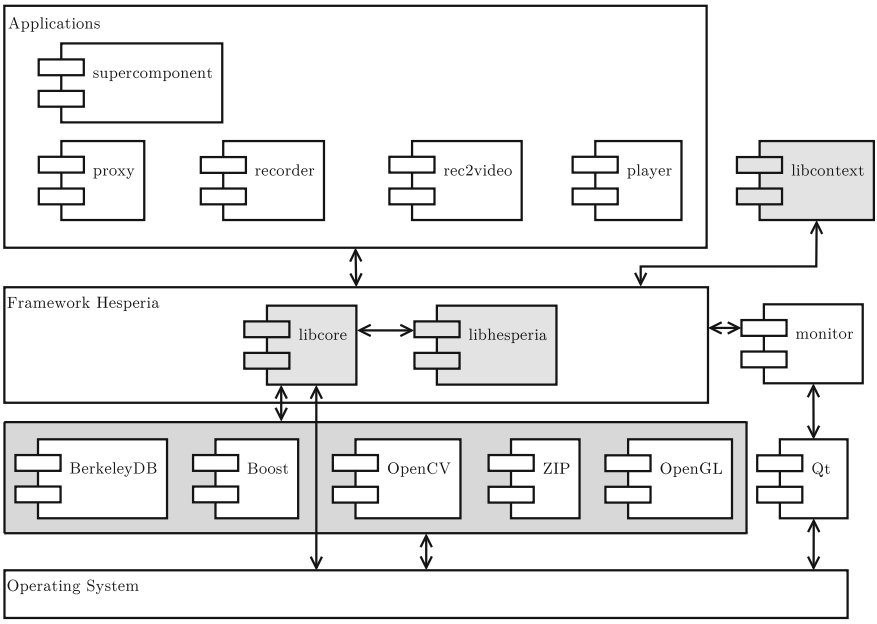


Fig. 9 Components of the Hesperia software framework

rules could be defined; even more, complex and detailed 3D models from popular 3D modeling applications could be simply added to a scenario. Thus, not only a realistic appearance for the user could be realized but these models were also used to generate realistic raw sensor data, for example, as described in the following.

6.3 The Hesperia Framework

In Fig. 9, all core components of the Hesperia software framework are shown. First, one of Hesperia's core concepts is to decouple a module's dependencies to third party supplied libraries. Therefore, inspired by the *design-by-contract* concept, only interfaces of a module's required existing algorithms are available for the module. The library supplied by a third party is entirely wrapped in lower layers allowing a transparent exchange if necessary. For example, for fast data storage, the Berkeley DB is transparently available to all modules; for processing images as well as matrices, the OpenCV library is wrapped and a generic interface is exported to higher layers.

In *libcore*, rudimentary system services are implemented and intensely tested using CxxTest. These services include concurrency for realtime and non-realtime services, I/O handling for URLs, and the basic communication concept *ClientConference*. A *ClientConference* is an enhanced wrapper around the UDP

multicast protocol. Using UDP multicast as the core communication concept, all data is sent only once regardless of how many recipients are listening.

Furthermore, the communication is realized using untyped "conferences" and typed messages called *Containers* which contain one serialized object. Using this concept, the receiver can use different thread-safe data storages like FIFOs, LIFOs, or key/value maps provided by *libcore* to receive data using the listener/observer pattern. Furthermore, these data storages can be simply reused to filter and combine different data like position data and features extracted from images.

The serialization of data structures is realized in a query-able manner. Therefore, a class' attributes-dependent identifier is computed at compile time. This identifier is used to access a class' attributes in a random manner during deserialization. Thus, changing the order or adding new attributes to a class do no break existing components; furthermore, these existing components do not need to be recompiled.

For supporting the development, some additional components are provided with Hesperia. The most important component is *supercomponent* which is mandatory for every *ClientConference*. This component tracks the lifecycle of each running component and delivers the module-dependent configuration data using the *dynamic module configuration protocol* (DMCP) which is inspired by the well-known DHCP. Thus, the configuration data can be modified centrally on the supercomponent's node and is consistent for a complex embedded system which is distributed automatically to several computing nodes.

As mentioned before, components for recording and replaying data are also available. These components can transparently record all transferred data of a running system without interfering with other components by simply joining a running *ClientConference*. For producing videos to illustrate a traffic situation, the component *rec2video* can be used. This component reads a previously recorded session and generates frames to render a video file.

6.4 Generating Synthetic Sensor Raw Data

As mentioned above, the Hesperia software framework provided libraries to generate synthetic raw sensor data. An example of results for a single layer laser scanner is shown in Fig. 10.

To produce this raw data, the formally specified system context was used, which was provided by the Hesperia software framework. Algorithms also described in [3] were integrated in the software framework which is outlined briefly in the following. The algorithms are based on a scene which can be rendered using OpenGL. Therefore, the system context is transformed automatically into a renderable scene representation using the visitor concept which traverses the ASG produced by the DSL's parser.

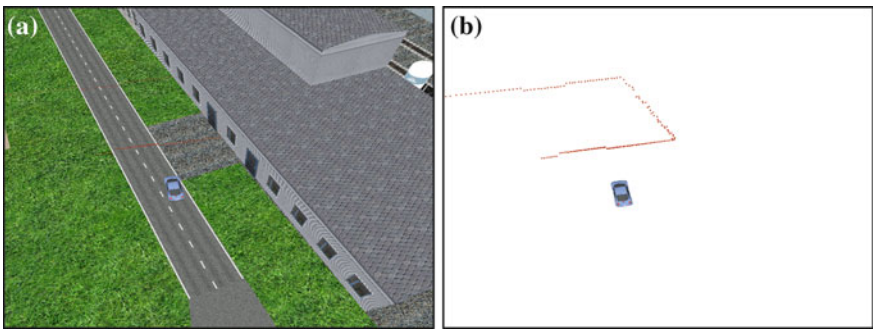


Fig. 10 The Hesperia software framework can be used to generate synthetic raw sensor data for a single layer laser scanner and camera systems. In this example, three independent scanners are defined: two pointing to 8 and 15 m in front of the vehicle, which are scanning orthogonally, and one scanner which is scanning in a parallel manner to the driving direction. **a** scene with three single layer laser scanners, **b** same scene with the hidden stationary context

After generating the scene graph, a camera, which is looking into this scene, is defined for each single layer laser scanner. This camera is positioned to the corresponding virtual mounting positions of the scanners.

Following, a so-called projector is created for each camera. This projector is used to project a texture into the scene which describes the shape of the scanning line. Using a specialized shader program which is executed on the GPU, the distances to the camera, where the line hits the first visible object, are encoded into the resulting rendered image.

The final step of this algorithm is to analyze this synthesized image by interpreting the encoded distances to transform them into the local scanner coordinate system. Then, the resulting coordinates can be serialized and distributed to imitate a sensor's protocol.

A similar principle without the shader programs can be used to simulate a simple color or gray-level camera. Therefore, the current rendered image "captured" by the OpenGL camera is read back from the GPU and provided as input data for vision algorithms.

6.5 Automating System Simulations

As mentioned before, *libcontext* and *libvehiclecontext* can be used to automate the execution of system simulations. As outlined in Sect. 5, the customer's needs must be available to define acceptance criteria for the software under test. Compared to unit tests, acceptance criteria define the set of conditions when to evaluate a test scenario as passed or failed.

The Hesperia software framework was successfully applied to a research and development project for the development of an autonomously driving vehicle

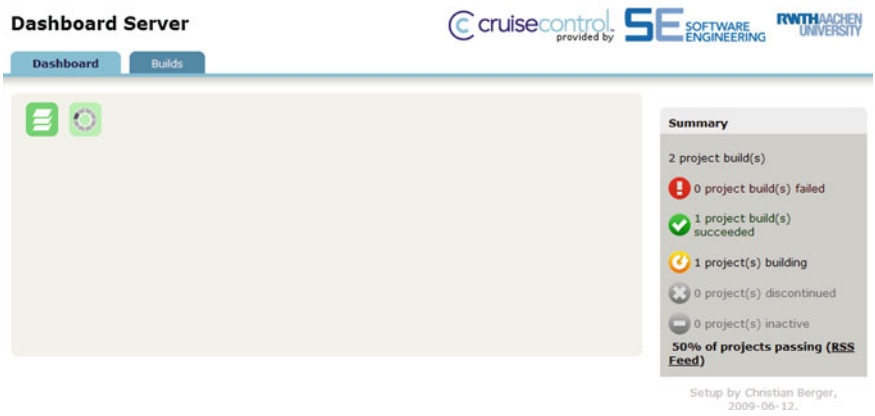


Fig. 11 Quality assurance with the continuous integration system CruiseControl. Using continuous integration, the same source code and test cases can be checked for different compiler options as shown in the figure, or even more for different platforms like Linux and FreeBSD. Thus, the developers can concentrate on the core development of the algorithms and only need to modify or correct the code if one integration server reports an error for a specific platform

which should navigate on a given digital map. The project was carried out at the University of California, Berkeley together with the RWTH Aachen University from June to August 2009. From this project three validators for continuously evaluating the software in a purely virtualized system's context were developed.

The first validator called *destination reached reporter* continuously supervises the current position and orientation of the vehicle. It returns true if the vehicle reaches its overall destination. This validator is registered at the virtualization layer at *libcontext* as a read-only listener for data sent in the *ClientConference*. Thus, the entire system does not need to be modified and can instead be inspected non-invasively.

The second validator is called *shortest route chosen reporter* and evaluates to true if the vehicle has chosen and driven the shortest route between two given waypoints. Therefore, it computes the shortest route using the digital map provided by the DSL which describes the stationary context. The result of this computation consists of a list of consecutive waypoints to be passed. During execution, this validator continuously supervises the vehicle's position and orientation and returns true when all waypoints are successfully passed.

The third validator, which is called *distance to route reporter*, continuously supervises the position and orientation of the vehicle and measures its distance to a given route. If the distance is never greater than a given threshold, this validator returns true.

All the aforementioned validators were combined in different test cases to define criteria for quality assurance using the unmodified CxxTest unit test framework; other validators are described in [3]. Thus, these test cases could be easily integrated with CruiseControl [7] as shown in Fig. 11.

Compared to the CarOLO project where a set of self-written scripts were used to realize a continuous integration system, CruiseControl could be applied without modification. Moreover, older test runs can still be accessed through CruiseControl's web front end for further inspection. For example, for two test cases to evaluate the aforementioned navigation algorithm there are 12 MB of results in XML that are generated per execution. Thus, appropriate post-processing, such as XSLT, must be applied to aggregate the information and to create a valuable report for the developers.

7 Related Work

The approach described in the first part of this chapter has been used to develop the autonomous driving vehicle "Caroline" for the 2007 DARPA Urban Challenge. In the CarOLO project a 2006 VW Passat station wagon was modified using car computers and many sensors for perceiving and understanding the environment and actuators for controlling the car by software. In [2], the simulation approach developed for the competition is presented. However, and as discussed in Sect. 6, the solution developed for "Caroline" had some drawbacks. Thus, as presented in the second part, the Hesperia software framework, written entirely from scratch, allowed a more convenient and repeatable way to design and execute system simulations [3]. Besides the results and experiences from the 2007 DARPA Urban Challenge, experiences from a research project to develop an automatically driving vehicle for highways formed the base for the development of Hesperia.

Compared to the approach presented in this chapter, similar approaches for simulation purposes are the CarMaker vehicle simulation by IPG Automotive GmbH and VEDYNA, [11] and [18], which use numerical simulations of full car dynamics with interfaces to MATLAB/Simulink. Both try to ease the development and integration of vehicle controllers. However, they do not support the consistent and integrated simulation of software architectures and software components using an embedded domain specific language as well as the re-use of available 3D models.

A similar approach to *libcore* in the Hesperia software framework is used in the automotive data and time triggered framework (ADTF) as outlined in [14]. The ADTF can be used to model a directed graph reflecting the data flow through a set of processing modules. The communication is realized using so-called channels, which themselves are typed but which can carry arbitrary typed data in principle contrary to the approach used in Hesperia which relies solely on typed messages. Furthermore, no support for a formally specified, consistent data model is provided.

In addition to the aforementioned ADTF, the toolkit virtual test drive is developed to manage previously recorded raw sensor data or to synthetically generate required input data to perform SiL-, HiL-, ViL-, or DiL-simulations [12]. Compared to the Hesperia software framework, an approach to generate raw sensor data for a single layer laser scanner, for example, is not available yet.

A similar approach to the Hesperia software framework and the tool-suite virtual test drive is used by TNO pre-scan [9]. This software can be used to support the development of so-called pre-collision driver assistance systems. Contrary to Hesperia, the synthetic generation of raw sensor data for a single layer laser scanner, for example, based on popular 3D models is not supported and no formal and integrated DSL is provided for the developer.

Another approach is provided by a tool from IAV [15]. This tool generates synthetic raw data for arbitrary sensors. The user models in a 2D manner the characteristics of a specific active sensor, like a field of view (FOV), a maximum distance, and some error noise. Then, the software computes preprocessed sensor data which would be provided by the ECUs of specific sensors. Contrary to the Hesperia software framework, only open-loop data generation is possible. Thus, no resulting sensor data in interaction with other vehicles can be generated. Furthermore, the sensors' models are only so-called visibility models which do not imitate the actual measurement principles.

8 Conclusion

Intelligent driving assistance functions need a detailed understanding of the vehicle's surroundings, of the driving situation, and the traffic rules and regulations as well as a sufficient knowledge about the physics of cars to fulfill their tasks. In the end, an intelligent driver assistant must be able to drive on its own. Thus, the 2007 DARPA Urban Challenge was a great opportunity to foster this area of autonomous vehicles and intelligent driving assistants. Developing this kind of complex software needs an innovative, agile development process that is compatible with the overall system consisting of a standard car, such as the VW Passat, sensors, actuators, and a number of computers suitable for automotive use.

For an efficient and stringent development project, a number of actions has to be taken, including an iterative development in small increments, early bug-detection and bug-fixing, stable version and configuration management, a solid architecture which embraces automated tests at any level of the software architecture, and most of all, a thoroughly designed test infrastructure. Evaluating a software's quality includes tests of the entire system, but for efficiency reasons it is important to test as much as possible while focusing on the subsystem under test. Thus, individual methods and classes are tested in the same manner as the whole reasoner. The test architecture allows us to fully extract the reasoner into virtual, simulated traffic situations, and allows checking the car behavior in various traffic situations efficiently. Automation of these tests allows us to (re-)run tests as desired at least every night or for every commit to the versioning system.

There are many complex traffic situations, let alone junction layouts and various possibilities of behaviors of other cars, so it was inevitable that there would be a need to run many tests in a simulated environment. The system simulation is rather general and will be usable for testing and interactive simulation in other contexts as well, for example being combined with HIL-tests.

The approach used and the results gained in the CarOLO project show that autonomous driving is still a few years ahead, but also that efficient development of complex software in combination with the overall system is possible if the development process is disciplined, yet responsible, agile, and assisted by appropriate modern tool infrastructures.

References

1. Basarke, C., Berger, C., Berger, K., Cornelsen, K., Doering, M., Effertz, J., Form, T., Gülke, T., Graefe, F., Hecker, P., Homeier, K., Klose, F., Lipski, C., Magnor, M., Morgenroth, J., Nothdurft, T., Ohl, S., Rauskolb, F.W., Rumpe, B., Schumacher, W., Wille, J.M., Wolf, L.: Team CarOLO–Technical paper. Informatik-Bericht 2008-07, Technische Universität Braunschweig (2008)
2. Basarke, C., Berger, C., Rumpe, B.: Software and systems engineering process and tools for the development of autonomous driving intelligence. J. Aerosp. Comput. Inf. Commun. **4**(12), 1158–1174 (2007). doi:http://dx.doi.org/10.2514/1.33453
3. Berger, C.: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles. No. 6 in Aachener Informatik-Berichte. Shaker Verlag, Aachen (2010)
4. Berger, C., Völkel, S., Rumpe, B.: Extensible validation framework for DSLs using MontiCore on the example of coding guidelines. In: Proceedings of the Symposium on Automotive/Avionics Systems Engineering (2009)
5. Boost: Boost::Spirit. http://spirit.sourceforge.net (2009)
6. Buehler, M., Iagnemma, K., Singh, S.: The 2005 DARPA Grand Challenge: The Great Robot Race. Springer Verlag, Berlin (2007)
7. CruiseControl: CruiseControl. http://cruisecontrol.sourceforge.net (2009)
8. Edgewall.org: Trac. http://trac.edgewall.org (2009)
9. Gietelink, O., Ploeg, J., Schutter, B.D., Verhaegen, M.: Testing advanced driver assistance systems for fault management with the VEHIL test facility. In: Proceedings of the 7th International Symposium on advanced vehicle control, pp. 579–584 (2004)
10. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: a framework for the development of textual domain specific languages. In: ICSE Companion '08: Companion of the 30th International Conference on Software Engineering, pp. 925–926. ACM, New York, USA (2008). doi:http://doi.acm.org/10.1145/1370175.1370190
11. IPG Automotive GmbH: IPG CarMaker. http://www.ipg.de/carmaker.html (2009)
12. von Neumann-Cosel, K., Dupuis, M., Weiss, C.: Virtual test drive—provision of a consistent tool-set for [D,H,S,V]-in-the-loop. In: Proceedings on driving simulation conference (2009)
13. Rauskolb, F.W., Berger, K., Lipski, C., Magnor, M., Cornelsen, K., Effertz, J., Form, T., Graefe, F., Ohl, S., Schumacher, W., Wille, J.M., Hecker, P., Nothdurft, T., Doering, M., Homeier, K., Morgenroth, J., Wolf, L., Basarke, C., Berger, C., Gülke, T., Klose, F., Rumpe, B.: Caroline: An autonomously driving vehicle for urban environments. J. Field Robotics **25**(9), 674–724 (2008). doi:http://dx.doi.org/10.1002/rob.v25:9
14. Schabenberger, R.: ADTF: framework for driver assistance and safety systems. In: VDI Wissensforum IWB GmbH (ed.) Integrierte Sicherheit und Fahrerassistenzsysteme, 2000, pp. 701–710. VDI-Gesellschaft Fahrzeug- und Verkehrstechnik, Düsseldorf (2007)
15. Schonlau, B.: Test und Absicherung von Funktionen mit synthetischen Umfeld- und Fahrzeugeigendaten. In: Gesamtzentrum für, Verkehr Braunschweig e.V. (eds.) AAET 2009—Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel, vol. 10, pp. 109–121 (2009)

16. Shipley, D., Celko, J., Walker, J.: DARPA announces urban challenge finalists. DARPA, Press Release (2007)
17. Sintara, N.: Entwurf und Realisierung eines plattformübergreifenden, graphischen Verkehrsszenario-Editors. Master's Thesis, Technische Universität Braunschweig (2008)
18. TESIS Gesellschaft für Technische Simulation und Software mbH: TESIS DYNAware. http://www.tesis.de/en/index.php?page=1004 (2009)
19. The Eclipse Foundation: Eclipse Rich Client Platform. http://www.eclipse.org/rcp (2009)
20. Tigris.org: CxxTest. http://cxxtest.tigris.org (2009)
21. Tigris.org: FSVS—Fast System VerSioning. http://fsvs.tigris.org (2009)
22. Walker, J., Jones, J.S.: DARPA announces third grand challenge: urban challenge moves to the city. DARPA, Press Release (2006)
23. Weiser, A., Bartels, A., Steinmeyer, S., Schultze, K., Musial, M., Weiß, K.: Intelligent Car—Teilautomatisches Fahren auf der Autobahn. In: Gesamtzentrum für Verkehr Braunschweig e.V. (ed.) AAET 2009—Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel, vol. 10, pp.11–26 (2009)

Empirical Evaluation of an Autonomous Vehicle in an Urban Environment

Ben Upcroft, Alexei Makarenko, Alex Brooks, Michael Moser, Alen Alempijevic, Ashod Donikian, Jonathan Sprinkle, William Uther and Robert Fitch

Abstract Operation in urban environments creates unique challenges for research in autonomous ground vehicles. Due to the presence of tall trees and buildings in close proximity to traversable areas, GPS outage is likely to be frequent and physical hazards pose real threats to autonomous systems. In this paper, we describe a novel autonomous platform developed by the Sydney–Berkeley driving

B. Upcroft (✉)
The University of Queensland, St Lucia, QLD 4072, Australia
e-mail: ben.upcroft@qut.edu.au

A. Makarenko · A. Brooks · M. Moser · R. Fitch
The University of Sydney, Sydney, NSW 2006, Australia
e-mail: a.makarenko@marathon-targets.com

A. Brooks
e-mail: a.brooks@marathon-robotics.com

M. Moser
e-mail: m.moser@marathon-robotics.com

R. Fitch
e-mail: rfitch@acfr.usyd.edu.au

A. Alempijevic · A. Donikian
The University of Technology, Sydney, NSW 2007, Australia
e-mail: a.alempijevic@cas.edu.au

A. Donikian
e-mail: a.donikian@cas.edu.au

J. Sprinkle
Electrical and Computer Engineering, The University of Arizona ,
Tucson, AZ 85721-0104, USA
e-mail: sprinkle@ece.arizona.edu

W. Uther
NICTA and The University of New South Wales, Kensington,
NSW 2052, Australia
e-mail: william.uther@nicta.com.au

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3_11, © Springer-Verlag London Limited 2012

team for entry into the 2007 DARPA Urban Challenge competition. We report empirical results analyzing the performance of the vehicle while navigating a 560 m test loop multiple times in an actual urban setting with severe GPS outage. We show that our system is robust against failure of global position estimates and can reliably traverse standard two-lane road networks using vision for localization. Finally, we discuss ongoing efforts in fusing vision data with other sensing modalities.

1 Introduction

The Sydney–Berkeley driving team is a collaboration between the University of Sydney, the University of California, Berkeley, the University of Technology Sydney, and National ICT Australia (NICTA). Our main goal in entering the 2007 DARPA urban challenge was to build a platform that could operate in an *actual* urban environment (Fig. 1a, b). Specifically, real-urban environments involve: (1) poor global positioning system coverage (GPS outage) (2) physical hazards such as buildings, trees, or other obstacles in close proximity to roads and (3) difficulties for vision systems due to changing lighting conditions. In addition, we were interested in addressing these challenges using relatively inexpensive and commonly available hardware and sensors. This paper describes our entry: a modified Toyota RAV4 (Fig. 2).

Our approach was to develop a robust system that could drive safely, minimizing unrecoverable errors. The overall strategy is to use local sensing to compensate for errors in global position estimation, and to use a hierarchical planning system to make intelligent high-level decisions.

Our system design incorporated many existing robotics techniques, including probabilistic state estimation, terrain modeling, sensor fusion, sample-based planning through the terrain map, and feedback control at multiple levels of abstraction. A novel aspect of the design compared with many robots from the previous DARPA Grand Challenges was vision-based localization using lane markings (Fig. 3a, b). Our test facility was particularly complex with GPS satellite acquisition denied over large sections (100s of meters) of the course. For this reason, a robust, local navigation solution through the combination of vision and dead reckoning was required.

One of the design requirements for the RAV4 was the use of a minimal set of relatively inexpensive sensors for traversal of an urban environment where global localization information was not always available. Using a single camera, a set of three 2-D laser range finders, a GPS receiver, and an inertial measurement unit (IMU), the RAV4 was successfully tested at speeds of up to $6\,\mathrm{m\,s^{-1}}$ (13 mph).

We conducted experiments in an urban field site near Berkeley, CA. The site had severe GPS outage due to tall trees and buildings, and also narrow lanes and frequent obstacles. Our performance in the DGC site visit was hindered by the

Fig. 1 **a** Our test facility: an *actual* urban environment. **b** The test facility as viewed from the vehicle

Fig. 2 The SBDT Toyota RAV4 autonomous vehicle

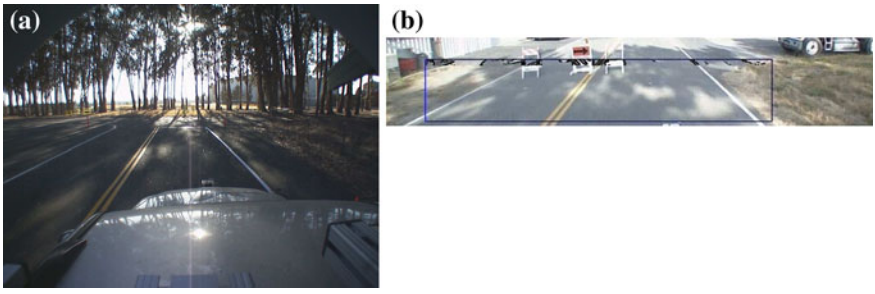


Fig. 3 Lane markings as viewed from the vehicle: **a** lanes in sun-dappled lighting, **b** lanes with obstacles

difficulty of the test course. However, in spite of this, we took the opportunity to make improvements and continue testing in this challenging environment, enabling us to conduct a significant empirical evaluation of our system. The results we report here show that the vehicle reliably navigates around an urban course

over 0.5 km in length. We were also able to perform high-level behaviors such as overtaking parked vehicles and proper queuing behavior approaching an intersection. We conclude that our vehicle is a robust platform for future research in autonomous vehicles, in particular in regard to ongoing research in fusing vision with other sensing modalities.

The chapter is organized as follows. Section 2 presents the hardware design of the vehicle. In Sect. 3 we describe the overall software system architecture, including algorithms and implementation. We report results from extensive field testing in Sect. 4, and conclude with a discussion in Sect. 5. We discuss related work throughout the text as is relevant. For collected related work we refer the reader to the special issues on previous grand challenges [13, 14, 18, 19].

2 The Vehicle Hardware

A stock 2006 Toyota RAV4 was used as our base vehicle. The lowest level of the control system consists of the vehicle actuation. This system was custom designed based on commercial systems while taking advantage of the existing drive-by-wire capabilities of the RAV4. Three components are critical to the control and operation of the vehicle: throttle, brake, and steering.

For the modifications to each of these components, reliability, actuation speed, and control accuracy were considered. In addition it was clear from the experiences in the previous DARPA Grand Challenge that rigorous testing was vital for success of the project. Finally, in case of electrical, mechanical, or software failure, an emergency stop over-ride automatically shuts down all autonomous actuation and brings the vehicle to a quick stop without the need for external commands. For autonomous testing, operations, and logistics in general, the vehicle is kept in a state which facilitates switching between robotic and human control without any need for mechanical alterations. Each actuation subsystem is controlled through independent switches accessible to a human observer sitting behind the front passenger seat. This allowed a great amount of flexibility for testing since some functionality of the system could be controlled by a human driver while other parts were completely autonomous. In addition, modifying the vehicle for normal driving and returning it to a state that is street legal can be performed within minutes.

2.1 Actuation

2.1.1 Steering

The RAV4 is equipped with electric power steering. In normal operation a sensor measures the torque applied to the steering column by the human driver. The sensor outputs an analog signal which the engine control unit (ECU) interprets as a

torque. The ECU then drives an assist motor which applies an assisting torque. In autonomous operation, the torque sensor signal is replaced by a simulated signal provided by an analog output card (Advantech PCI-1721) installed on the QNX onboard control computer. Care was taken to ensure that the simulated signal was close to the expected signal so that it was accepted as genuine and fail-safe states were rarely triggered.

This modification allowed the vehicle's built-in safety features (stability control, damping of too aggressive inputs) to remain in operation. It also allowed the car to returned to the original (street legal) configuration by disconnecting the custom control and connecting the original torque sensor.

2.1.2 Throttle

Throttle control in the RAV4 is achieved with an analog position sensor located in the accelerator pedal. It provides a signal to the throttle in a similar manner to that of the torque sensor for steering. As for the steering, the connector to the sensor is replaced with a custom adapter for control from our analog output card.

2.1.3 Brakes

Unlike the previous actuators, the brake does not have an existing electronic interface and thus required retrofitted actuation. In the previous grand challenge there were three main choices for brake actuation: linear electric actuators (used in the majority of vehicles for DARPA grand challenge, 2005), (electro-) hydraulic actuators, and pneumatic actuators (a sizable minority, 2005). Linear electric actuators allow good position control. However, their major drawbacks are relatively slow speed and the lack of force control. Although hydraulic systems provide good position and force control, we dismissed this option in the early stages since there was little relevant experience in the use of these systems within the team.

We chose a pneumatic system as it is fast, allows good force control, and we had experience within the team using these systems. Note that one drawback with pneumatic actuators is that they are limited in position control. We also desired the ability for parallel brake actuation by a robotic and human driver as described in several designs for the 2005 grand challenge.

To avoid space constraints in the driver's seat, the vehicle was converted to dual brake controls (as in driver training vehicles) and the front passenger seat removed. This conversion was performed by a certified mechanic ensuring the vehicle remained street legal. The brake actuator could then be mounted to the existing strong points designed for the seat and attached to the dual brake guide cable (which would normally connect to an additional brake pedal). Note that for manual driving, the guide cable is mechanically detached from the actuator. The pneumatic cylinder is controlled using the same analog output card as

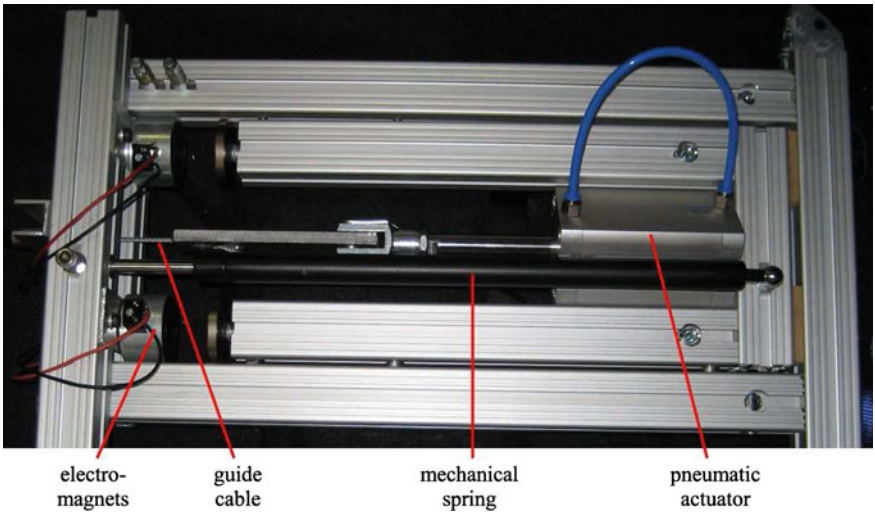


Fig. 4 Pneumatic brake actuator and failsafe system

the other actuators. The cylinder pulls the guide cable longitudinally to apply the brakes.

To ensure a fail-safe system, we adopted the approach of the Austin Robot Technology Team from the 2005 DARPA Grand Challenge [4]. The *U*-shaped assembly shown in Fig. 4 can move relative to the rest of the frame. The system is shown in fail-safe mode (maximum brake applied) with the mechanical spring pushing the sled to the right and hence applying force on the brake. To reset the system from the fail-safe state, the sled is manually pushed forward, with assistance from the pneumatic actuator, closing the gap between the sled and the electromagnets. If power is supplied to the electromagnets, they hold the sled in the left position with the brakes released, giving the pneumatic actuator full control. This setup is failsafe in the sense that loss of power will result in the electromagnets releasing the sled and the mechanical spring applying steady braking force.

2.2 E-Stop

The primary concern of the *e*-stop system is the safety not only of the team members but also of the general public, with an additional concern of ensuring that damage to property and equipment does not occur. Furthermore, it was required that the vehicle be maintained in a manner in which it can be switched back to a street legal condition. Thus, the *e*-stop system implementation and control of the actuators should never be allowed to interfere with normal driving on public roads. To guarantee zero interference during normal driving the *e*-stop system and

actuator control circuitry are physically disconnected from the vehicle controls ensuring that the two systems are isolated. The current wireless *e*-stop (Hetronic ERGO V4, operation frequency = 434 MHz) is designed to be easily interchangeable with the DARPA provided e-stop via a single connector.

To maintain simplicity, our implementation of the e-stop system is limited to the three *e*-stop modes as specified by the DARPA guidelines: *disable, pause,* and *run* [8]. For safety and reliability it was decided to implement the *e*-stop modes and their transition completely in hardware. This prevents potential software bugs from resulting in an unknown condition or state of operation.

The *disable* mode was implemented by removing power from the electromagnets. *Pause* mode supplied a hardware-tunable input to the pneumatic actuator, bringing the vehicle to a controlled stop. *Run* mode connected the pneumatic actuator to the analogue output from our computer.

2.3 Actuator Controllers

Since the trajectory planning is handled at the motion planning level, our low-level control interface consists primarily of vehicle speed (throttle and brake) and steering angle, both executed using proportional-integral-derivative (PID) feedback loops. The controllers also consider secondary state information (e.g. which gear is currently active, and car error states).

2.3.1 State Information

Using the state information provided by the vehicle's controller area network (CAN) bus and the inertial navigation system, we were able to accurately characterize the car's parameters. Most importantly, actuator positions could be mapped to actual states such as acceleration/deceleration, speed, and steering angle for precise low-level control.

We are using an opto-isolated CAN-controller card by PEAK system (PCAN-PCI) to interface with the vehicle's CAN-bus. This card can be used under the real-time operating system QNX, using a third-party driver by bitctrl systems. With this setup we achieve high data rates (500 kb/s for the vehicle's CAN-bus), high frequency (up to 80 Hz, depending on message priority), and excellent timing accuracy.

2.3.2 Steering Controller

Steering was the simplest controller and just required a PID feedback loop which corrected the error in steering angle. The built-in power steering motor provides the capability to turn the wheels from lock-to-lock in 1.2 s. However, the PID

gains that we are currently using have been tuned for smooth operation and take approximately 2 s.

2.3.3 Speed Controller

The speed controller consists of two PID controllers (brake and accelerator) and an arbiter to choose the controller for a given situation. The average speed of the rear wheels (calculated from the wheel encoder information on the vehicle's CAN-bus) provides the feedback with corrections at a frequency of 80 Hz. The arbiter allows only one controller to run at the same time; braking and accelerating cannot occur simultaneously. It also considers the vehicle state, so that the maximum RPM cannot be exceeded and the throttle can only be applied while in gear.

Control of the brake was more difficult than the throttle due to static friction in the mechanical system. As a result, brake force was not proportional to the applied actuator pressure. Instead, the effective braking force would remain relatively constant and at irregular intervals jump to match the actuator force. Further testing showed that small variations in actuator force generally resulted in unpredictable and jumpy responses in effective braking force. However, step inputs of varying size resulted in accurate proportional brake force response.

To solve the static friction problems in the control system, we quantize the output of the brake controller. Whenever the output of the controller is commanded to change, the actuator is set to zero force for 60 ms and then returns to the quantized set point. This ensures the effective brake force matches the controller's output by transforming a small correction into a large one and eliminating the effects of static friction.

2.4 Computing and Networking

The onboard computer system currently has four hosts (not counting diagnostic laptops that are often connected to the system). The onboard computers use two operating systems: Linux (Kubuntu) and QNX. The computing hardware consists of off-the-shelf rack-mounted PCs with Intel dual-core processors. The hosts are connected with a standard 1-Gigabit ethernet hub. Latency performance tests with this setup are shown in Fig. 5.

Some, but not all, parts of our system require real-time features. For example, there is a strong need for accurate timestamping of sensor data. A vehicle in the competition moves at speeds of up to 30 mph (48 kph) and small sporadic delays in the standard Linux kernel can have significant negative impact, particularly in navigation. To address this issue, we use a dedicated host running the real-time QNX neutrino operating system and one with the preemptive linux kernel patch for all interactions with sensor and actuator hardware.

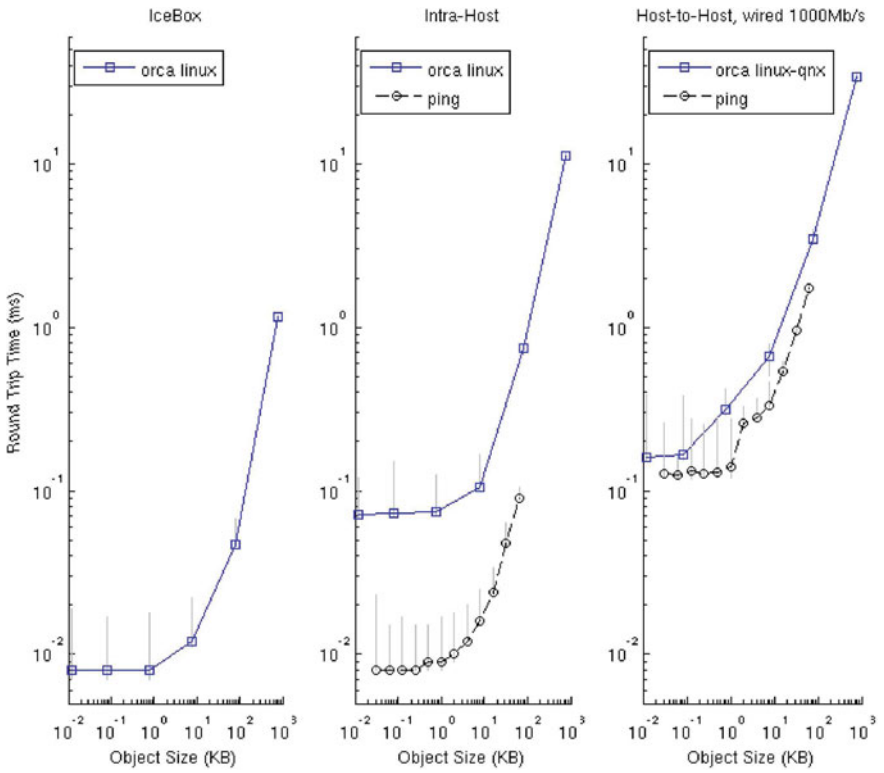


Fig. 5 Communication latency in our on-board computing system. We measured round trip time (RTT) for sending small to medium size messages within and between hosts. The plots show average RTT for sending 100 objects with an interval of 250 ms. Vertical bars show minimum and maximum values. Ping results are shown for comparison

2.5 Sensors

Perception of the environment and the vehicle state is performed by on-board sensors. Most of the vehicle sensors have been mounted on the custom built roof rack (Maytec rails) shown in Fig. 6.

Visibility of the terrain is greatest on the roof and obstruction of signals received by the GPS is at a minimum in this position. Fusion of GPS and output from an inertial measurement unit (IMU) provides the vehicle navigation solution. The IMU is mounted to the chassis above the transaxle. Currently, perception of the environment is performed using two SICK LMS291 lasers and a single camera (Hitachi HV-F31F, 3CCD progressive scan). One laser is pointed forward at a pitch of 7°. This laser records the cross-section of the terrain at a range of 16 m in front of the vehicle. The second laser is mounted vertically and is used for estimating the relative pitch of the road out to approximately 25 m depending on the reflectivity of the road. The lasers are connected to a Bluestorm/Express

Fig. 6 The sensor suite: SICK lasers, INS, and camera

PCI-Express 4-port serial card and communicate using RS-422 in order to run at a 500 k baudrate. The clock onboard the serial card was replaced with a clock that could provide this non-standard speed.

The camera is used for detection of lane markings out to approximately 40 m limited by resolution of the camera. It is connected to the computers via FireWire.

Estimating the vehicle state accurately is a key requirement for sensible path planning and representation of obstacles. Errors in the pose estimate can result in poor terrain maps with phantom obstacles leading to seemingly poor control decisions. Thus it is important that the navigation solution, at least at the local level, is smooth and locally accurate. Navigation is the central sensor system (Novatel ProPak-V3 GPS receiver combined with a Honeywell HG-1700). We use OmniSTAR HP global GPS corrections. All other sensors are registered with the navigation subsystem and fused using this information.

3 System Architecture

The urban grand challenge not only poses the problem of autonomy at the hardware level but also at abstract levels such as perception and decision making usually only performed by human operators. The SBDT has chosen a classic N-tier architecture [1] to address the large range of abstractions required in this competition (Fig. 7). The tiers, separated by the level of abstraction in control and perception, include modeling, planning, and execution elements.

The N-tier architecture encodes the following separation of concerns. The task of driving from point A to B is first considered on the global level. Given the town map a trip plan is generated in terms of GPS-referenced waypoints. Travel between waypoints can only be performed by following streets (ignoring parking lots for the moment). Perception, modeling, and planning motion along a street segment in the presence of traffic and complying with traffic rules is handled by the street level of the architecture.

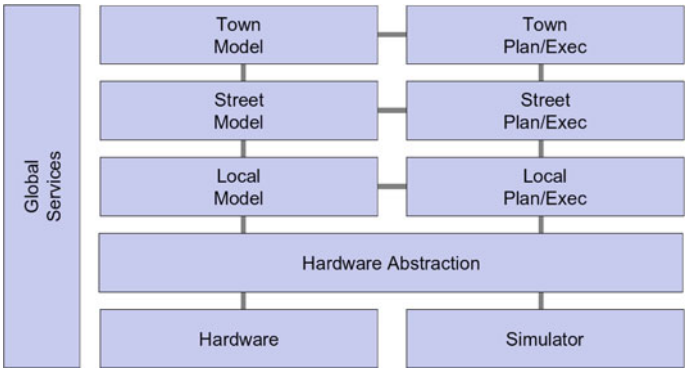


Fig. 7 The SBDT system architecture uses a classic hierarchical approach

Local environment modeling and action planning guarantee physical safety of the vehicle as it moves toward its goal. Information about the local environment and the vehicle itself is extracted from on-board sensors. The extent of modeling and planning at this level is limited by the sensor range and a finite (short) history of past observations. Some of the representative subtasks at each of these levels include tracking the vehicle location relative to the map: identifying street boundaries, buildings and other cars, controlling vehicle motion, reasoning about traffic rules, and many others.

3.1 Software Architecture

Building reliable robotic hardware is a difficult task in itself, but the major challenge of this competition is in software. The main difficulties arise from the task's complexity and its dynamic real-time nature. Other contributing factors include the distributed and cross-platform computing environment, the large number of software contributors, and the need to reuse existing code.

Our software is built using a component-based software engineering (CBSE) approach. This offers modularity, software reuse, and flexibility in deployment, all of which are necessary to address the problems listed above. Applied to a robotic application, CBSE means that algorithms for perception and navigation are mapped to a set of components. Components run asynchronously and exchange information through communication.

We use ZeroC's Ice middleware[1] extensively in our system: for component interface definition, inter-component communication, component deployment, location, activation services, etc. We also use Orca,[2] an open source project that

[1] http://zeroc.com/

[2] http://orca-robotics.sourceforge.net/

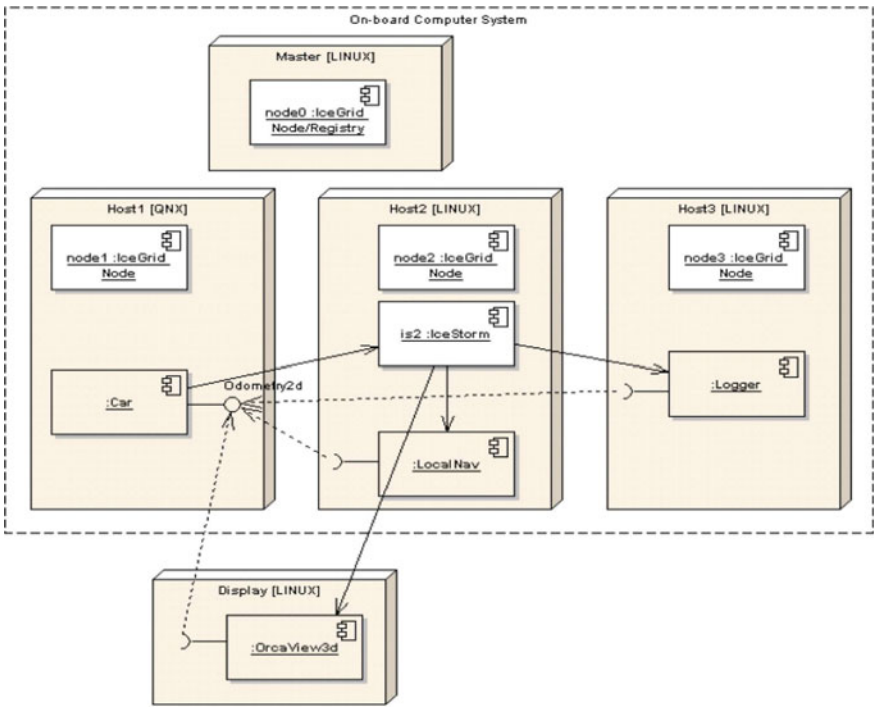


Fig. 8 Deployment diagram illustrating how the onboard computer system is configured

customizes ice to robotic applications and provides an online repository of reusable components.

The total number of components that comprise our current system is <20. However, this is expected to increase as the complexity and number of algorithms increase (for reference, the winner of the 2005 DARPA grand challenge had approximately 30). The software is written by about a dozen people from four organizations (this number is higher if we count the authors of the existing components used directly in our system).

Figure 8 illustrates the current deployment strategy.

The master host executes an instance of an icegrid node with a collocated icegrid registry. Every other host runs an icegrid node. One host runs QNX Neutrino, the remaining hosts run Kubuntu Linux. The QNX host executes low-level actuator and sensor components (our own partial, unsupported port of ice to QNX is available through the ZeroC developer forums). For brevity, not all system components (e.g., lasers) are shown. Some components that we use in the vehicle are currently publicly available, such as the laser range finder, the inertial navigation system, and others; due to the competitive nature of this project, other components, such as the car component that drives the vehicle are not publicly available. However, we intend to release most of these components through the Orca project in the future. Using Ice's tools we can manage the

system from a centralized XML file, and distribute the system's components to computational nodes.

3.2 Levels of Abstraction

The levels of abstraction are reflected in the system architecture (Fig. 7). We use a layered approach to planning and control. Our approach consists of a town planner, a street-level planner, and a local-motion planner. This layering achieves a number of benefits. First, it is more efficient to plan in a layered manner. We only plan in high detail through a region about which we have information. Also, it is an important design principle of the team's entry that driving is performed at a local level. We do not need to know where we are in a global space to retain safe control of the vehicle. Finally, although both the town and motion planners are in metric spaces, the layering allows these metric spaces to be decoupled. This enables us to use two navigation solutions: one smooth, but perhaps not globally accurate (from the IMU and camera), and another one globally accurate, but usually not smooth (from the GPS unit). The local obstacle detection and tracking uses the smooth navigation solution, and the global positioning uses the globally accurate positioning. We describe the three layers of our planning system (as shown in Fig. 7) in this section. Perception and planning will be considered together for each level.

3.3 Town Level

The route planner reads in the route network definition file (RNDF) and mission definition file (MDF) supplied to the team, and uses them to plan a high-level path that fulfills the mission defined in the MDF. On startup, the RNDF is processed to produce a connected graph (Fig. 9).

While driving, this graph is searched each second to discover a route to the next checkpoint, and then the route is processed to determine where the vehicle should next stop or exit the current street. That next exit or stopping point is passed to the street-level planner, along with any available information regarding the current road segment such as speed limits, estimated Euclidean distance to the exit point, and road curvature. The algorithm is summarized in pseudocode as Algorithm 1.

Algorithm 1 Town-level route planning

```
1: Preprocess RNDF to build graph G
2: while running do
3:     Localize robot in G
4:     Get obstacle (blockage) data from local perception level
5:     Add blockages to G
6:     Search G for best path to next checkpoint
7:     Send initial path segment to street-level planner
8: end while
```

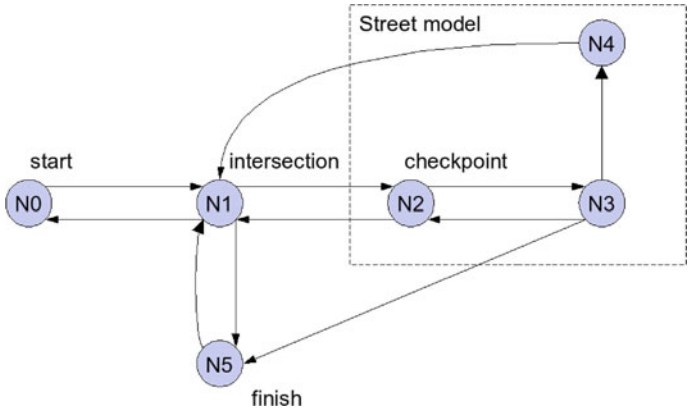


Fig. 9 Town model—a connected graph constructed from the RNDF

Once the graph is constructed, we find paths using the standard A^* search algorithm [12]. In this search, links have a cost proportional to their length, lane change links have a linear penalty, and exits also have a small penalty. The search starts at the closest location in the RNDF to our current global position, as supplied by our GPS/INS unit, and runs to the next checkpoint the vehicle is required to pass through. We then traverse the resulting path to find the first stop sign or exit.

During a run, it is possible to modify the graph in two ways. New waypoints can be added, increasing the accuracy of the RNDF. Also, lanes can be marked as blocked (using information from the local perception level), which would cause a different plan to be found during subsequent iterations.

3.4 Street Level

The street-level planner is the component in our system that plans for lane changes and intersections. The role of this layer is to plan ahead along the current street, avoiding large obstacles such as cars, until the next stop sign or exit (which was passed down by the route planner). Plans are constructed in a 1.5 dimensional, discrete space. That space has distance down the current segment in 3m increments as one-dimension, and lane number as a second dimension. Each discrete cell is either free or contains an obstacle passed up from the local level. Every obstacle is assumed to have a discrete velocity along the road, allowing our street-level plan to anticipate the actions of other vehicles. The town map passes in the rough shape of the road ahead in terms of the curvature of each segment. This can be used to reduce speed ahead of sharp corners. Figure 10 illustrates the representation. Planning proceeds with a simple model of other vehicles using real-time dynamic programming [2].

The local motion planner in our system takes commands in the form of a series of waypoints specifying a trajectory. The street-level planner gives commands in

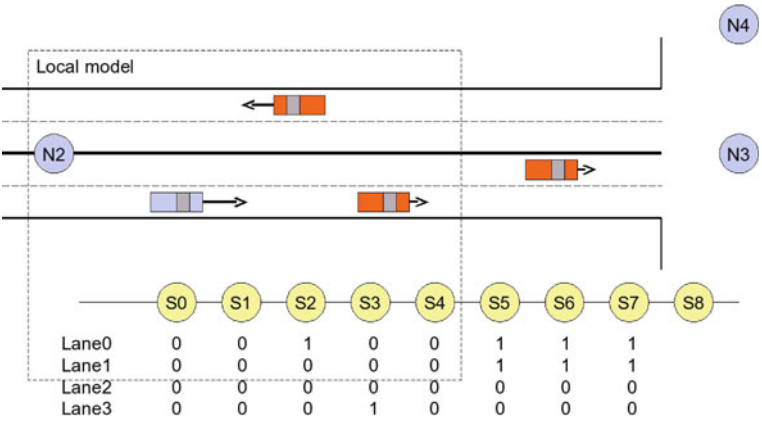


Fig. 10 Street model: a 1.5-dimensional, discrete space with distance down the current segment in three meter increments as one-dimension, and lane number as the extra half dimension

terms of where to change lanes and a maximum speed. We have a simple system that maps between these two types of commands.

The mapping takes the lane center-line generated from the town-level RNDF map, and the detected lane information. The map center-line is shifted laterally until it is aligned with the lanes detected by the car's vision system. We then loop along this corrected center-line until the distance for the next street-level command is reached, passing the center-line down as the trajectory, and marking the maximum speeds as appropriate. If a lane change is detected then the trajectory is offset as appropriate to place it in the correct lane. Figure 11 illustrates the waypoints generated by the street-level planner during a lane change.

At a corner, there is no trajectory information passed down. Rather, the street-level planner provides the low-level planner with the next waypoint in the town-level map, and the low-level planner calculates a smooth trajectory for the turn.

3.5 Local Level

3.5.1 Static Obstacle Detection and Representation

For reliability and robustness, we have chosen to use a standard grid map data structure often used in robotics to represent obstacles in the environment [11, 23]. We closely follow the obstacle detection technique used by Thrun et al. [23] in the last challenge, in which the scans from downwards looking lasers are projected into the vehicle's local coordinate frame, resulting in a 3-D point-cloud for each laser. Laser points are grouped into 2-D cells which combine to make a surface grid. The traversability of a cell is classified by vertical height differences between laser points within a cell. A cell with a height difference greater than some threshold is deemed not traversable. Unlike Thrun et al. [23], we did not need the

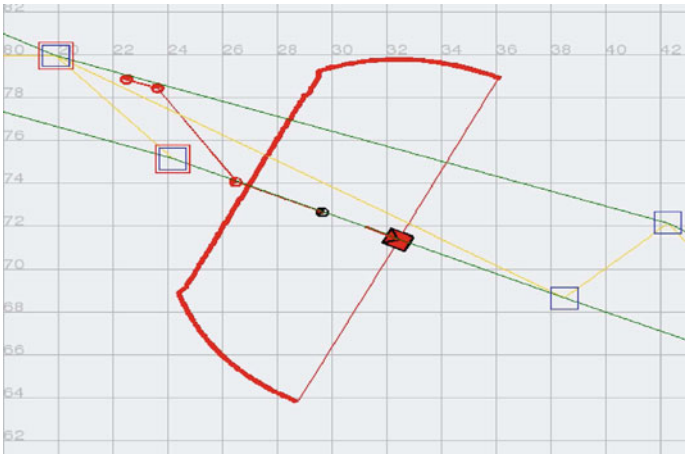


Fig. 11 Screen capture of our visualization tool as the car approaches a lane change. The *blue boxes* are RNDF waypoints. The *red boxes* are RNDF waypoints that the car plans to travel through. The *small red circles* are low-level waypoints for the local navigation component to follow. The *large red D* shape is the laser return. Note that once the local waypoints are past the last RNDF waypoint in the current lane, they are offset into the neighboring lane to cause the car to change lanes

additional machine learning layer required to compensate for small inaccuracies in pose. We found that accurate timestamps recorded using the hard real-time QNX operating system for both the navigation solution and laser scans resulted in no such error.

An example of a traversability map is shown in Fig. 12. This map is a display version of the grid's data structure which is provided to the navigation and path-planning components.

3.5.2 Lane Detection

Unlike the first two grand challenges, abstract objects such as lane markings must be detected and stored for use in high-level decision-making. Although reflective paint can assist lasers in detecting lane markings, we have selected to use visual sensors in case poor or non-reflective lane markings were present. Detection of lane markings in image data is not a trivial task. The difficulty is mainly due to the lack of unique models, poor quality of lane markings due to wear, occlusions due to the presence of traffic, and complex road geometry.

The potential pitfalls when using image data led us to the conclusion that any method using a strong model of the road (lane) will eventually fail, thus, weak models must be applied. Lane detection is generally based on the localization of a specific pattern (the lane markings) in the acquired image and can be performed with the analysis of a single still image.

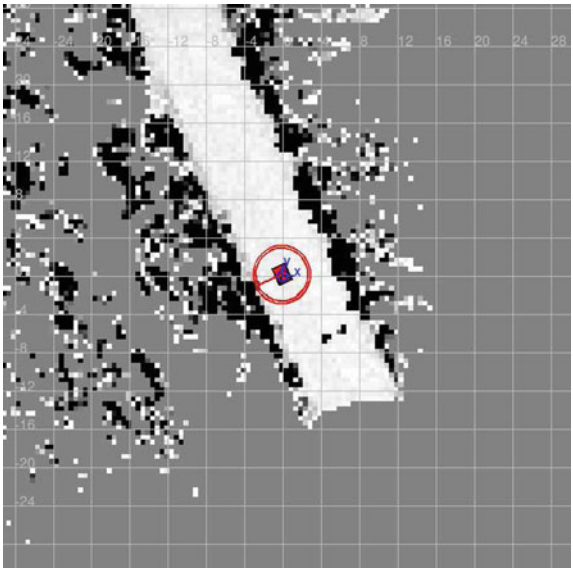


Fig. 12 Snapshot from the GUI software component illustrating the traversability map. White indicates traversable, *black* indicates an obstacle larger than 30 cm, and shades of *gray* indicate objects 0–30 cm. Cell sizes are 40 cm. The *red rectangle* represents the vehicle

The method for lane marking detection employed in our paper is based on the generic obstacle and lane detection (GOLD) implementation [3]. In order to fit the lane model to the acquired road images, geometrical image warping is performed with inverse perspective mapping (IPM). The IPM technique projects each pixel of a 2-D perspective view of a 3-D object and remaps it to a new position, constructing a new image on an inverse 2-D plane. Conversely, this will result in a bird's-eye view of the road, removing the perspective effect. Each pixel represents the same portion of the road, allowing a homogeneous distribution of the information among all image pixels. To remove the perspective effect, it is necessary to know a priori the specific acquisition conditions (camera position, orientation, optics) and the scene represented in the image (the road). The mapping assumes that the road is flat. This assumption does not hold in most cases, so we use a vertically-mounted laser to detect the relative pitch of the camera and the road.

The resolution of the remapped image has been chosen as a trade-off between information loss and processing time. At the moment we are using the red channel of an RGB image as it provides the highest contrast between the lane and pavement color. The approach was further chosen, as the camera selected on our vehicle is a 3CCD camera, enabling retrieval of each image channel without any compression losses from a Bayer pattern. Another potential image space to be used is the C1C2C3 space [21], which provided marginally better results but at the cost of computational effort.

Lane marking detection is performed on the re-mapped image by applying a lane model which stipulates that a road marking is represented by a predominantly vertical bright line (lane marking) of constant width surrounded by a darker region (the road). Thus, the pixels belonging to a road marking have a brightness value

Fig. 13 **a** Original image. **b** IPM image. **c** Detected lane marking segments

higher than their left and right neighbors at a given horizontal distance. A vertical edge in an image conforms similarly to the same principle; the intensity difference between neighboring pixels must be over a threshold to be validated. This is detected by performing a sobel edge operation [10] on the red channel of the image. An exhaustive search across each row of the image will produce potential lane marking candidates where the match probability can be measured with the edge quality, i.e., the difference in intensity.

It is assumed that the center lane marking consists of a double yellow line and the other lane markings consist of a single white line. Rows in an image are sequentially analyzed. Pixels within a row that are classified as part of a lane marking form an "observation" for a spatial, recursive filter which propagates vertically through the image. A pixel is deemed an observation if there are similar adjacent pixels that conform to a fixed lane marking width when grouped together. Propagation of the filter occurs using a transition model which assumes that lane markings are more likely to have a low degree of curvature and that there is a maximum degree of curvature which cannot be exceeded. An example of detected lanes is shown in Figs. 13a–c and 14.

Finally, the virtual lane trajectory is calculated from the detected center lane marking additionally verified against the other lane markings (width of lane defined approximately by the RNDF) and represented as a polyline. It is then communicated to the path-planner and incorporated as a constraint. The validity of this approach is demonstrated in Fig. 15a–c. The detected lane marking is represented as a thick polyline noted several meters in front of the vehicle. The lane marking is superimposed on a traversability map and accompanied with lane markings (noted as thin black lines) generated from the supplied RNDF and observed global navigation solution. The detected lane marking is not incorporated as a constraint in Fig. 15b, whereas the lane marking constraints the generated virtual lane trajectory in Fig. 15c.

The detected center lane marking is propagated between the image frames using the integrated solution of the onboard inertial measurement unit. The position of the propagated center lane marking is then used as a seeding point for the next iteration of the filter and contributes to the robustness and improved efficiency of the algorithm. Mean processing time for detecting the initial lane trajectory is

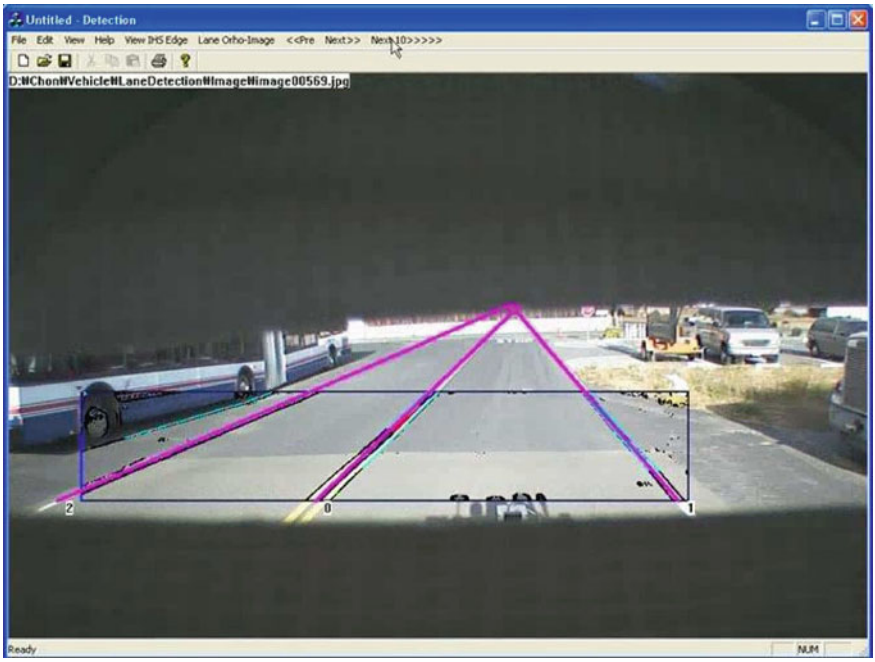


Fig. 14 Detected lanes: *pink lines* represent the observed lanes projected onto the image in perspective. The *box* represents the area of the image in which we search for lanes

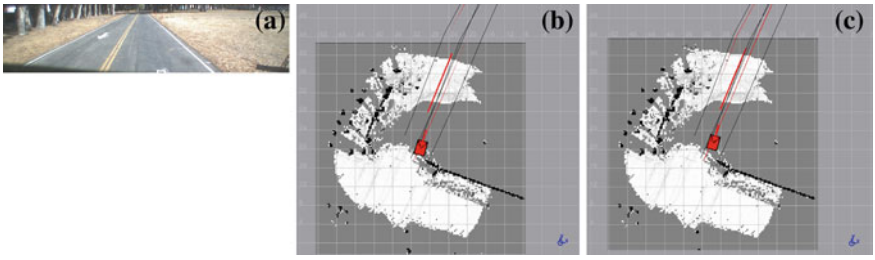


Fig. 15 **a** Original image. **b** Lane marking not incorporated into constraints by high level planner. **c** Inclusion of constraints by high level planner and new virtual lane trajectory

70 ms, but once the lane is detected and propagated between frames the processing time drops to 5 ms.

3.5.3 Local-Motion Planning

The trajectory and speed markers generated by the street-planning level provide goal information to the final component of our planning system, the low-level motion planner. The speed markers are treated as point goals by the system.

The responsibility of this component is to control the car to visit the goal waypoint sequence while avoiding obstacles and staying close to the provided trajectory (which represents the lane information).

We consider this task as an instance of the *kinodynamic motion planning* problem [9, 17]. In this formulation, the task is to find a collision-free path to the goal in the vehicle's state-space (configuration-space plus velocity parameters). Such a path represents an open-loop trajectory that drives the robot towards its goal. We applied a modified version of the planning algorithm proposed by Brooks et al. [5], which is closely related to the popular rapidly-exploring random trees (RRT) algorithm [7, 16]. RRTs have been used successfully with real robots and demonstrate acceptable real-time performance [6, 15]. We modified the basic algorithm to incorporate lane following and to respect velocity and acceleration bounds of our vehicle. The algorithm is summarized in pseudocode as Algorithm 2.

Algorithm 2 RRT-based motion planning

```
1: while running do
2:     Get new waypoints
3:     if no waypoints then
4:         Send stop action to actuator controllers
5:     end if
6:     Get obstacle data
7:     Get trajectory data
8:     Initialize tree to previous best path
9:     for i = 1 to desired number of expansions do
10:        With probability p:
11:            Randomly choose node n from tree
12:            Randomly choose safe control values
13:            Predict new state s
14:        With probability 1 − p:
15:            Randomly choose node n from tree
16:            Use heuristics to choose safe control values that guide robot toward goal
17:            Predict new state s
18:        Compute distance to nearest obstacle from s
19:        Compute weighted cost of leaf-to-root path from s
20:        Compute both optimistic and pessimistic cost estimates to the next goal
21:        if the total optimistic cost for this node is greater than the least pessimistic cost for any
           node then
22:            Prune path
23:        else
24:            Add s to tree
25:        end if
26:    end for
27:    Store best-cost path
28:    Send initial control action in best path to actuator controllers
29: end while
```

The standard RRT algorithm searches by iteratively expanding a search tree until it finds the goal. Here the start state is current robot position, and goal states are waypoints provided by the street-level planner. We expand the tree by forward integration of control values from a randomly-chosen node in the tree. The control

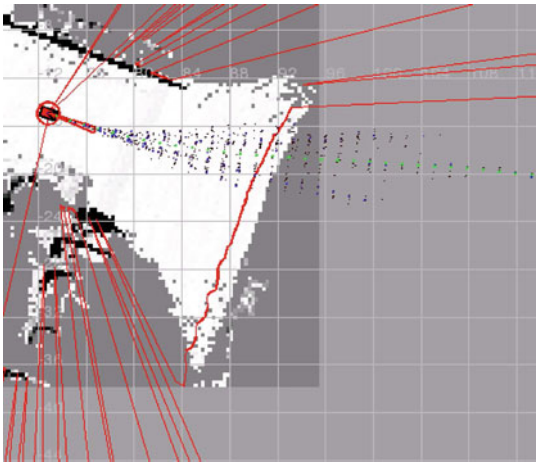


Fig. 16 The motion planner execution in our GUI. The vehicle is drawn as a *rectangle*, search tree nodes are drawn as pixels emanating from the vehicle. The *red lines* represents returns from one of the lasers projected in 2D

values respect velocity and acceleration bounds and are either determined by heuristics that lead toward the goal or are chosen randomly. We prune unfavorable leaves in the tree using a weighted-cost function that takes into account Euclidean distance to goal, distance to obstacles, proximity to the supplied trajectory, and path length. Because of this pruning, sampling is biased toward favorable regions of state-space. Obstacles are sensed using the laser scanner and provided to the planner in a grid representation.

The algorithm performs a fixed number of expansions and then returns the best-valued control action (desired speed, desired steering angle), which is then executed by the low-level hardware controllers. We interleave this planning with a check for new waypoints and current sensor data. Since waypoints do not generally change radically between calls to the planner, we store the last best path for use in initializing the tree in the next iteration. In fact, the vehicle rarely reaches any given waypoint before the street-level planner provides a new set (receding-horizon control).

We evaluated the algorithm's performance in (physics-based) simulation using the Player/Gazebo environment.[3] The CPU time spent choosing a control action ranged from 20 to 100 ms on a standard workstation running both the simulator and the planner. Similar performance was observed during online hardware testing with the vehicle. Figure 16 shows a screenshot of the GUI we used to visualize algorithm execution.

[3] http://playerstage.sourceforge.net/

4 Results and Analysis

To evaluate our system, we performed experiments in a real-urban environment. Our objective was to assess the robustness of our system in unfavorable GPS conditions. To do this, we performed navigation runs around a 560-m loop flanked by buildings, large trees, ditches, and other obstacles. We also demonstrated high-level behaviors such as overtaking parked vehicles and proper queuing behavior approaching an intersection. An aerial view of this test track is shown in Fig. 17.

We report results from two navigation runs, alternating travel direction (clockwise and counter-clockwise) around the loop. The runs consisted of ten laps each and evaluated the complete system including lane detection. Each run used the following protocol. We began with the system in *pause* mode at a common starting point (see Fig. 17g) and loaded a mission definition file (MDF) that encoded the desired travel direction and number of laps. We manually switched the system into *run* mode and continued until a failure requiring manual intervention. After a failure, we manually switched the system to *pause* mode, corrected the failure, and switched back to *run* mode. We continued in this way until we completed the desired number of laps. Raw data from all sensors (including camera images) was stored in a log file for analysis. Failures consisted of: (1) a hardware failure in the power steering system requiring turning the car's engine off, then back on (2) a false obstacle requiring us to manually drive the car past (3) a bad turn requiring us to manually drive the car back onto the road, or (4) a planning failure where the car failed to make a turn and was unable to correct itself via a *U*-turn.

We provide a summary of our results in Table 1. The majority of errors were due to hardware problems (power steering failures) or false positives in our obstacle detection subsystem. We had only one unrecoverable failure due to a bad turn, and one due to a planning error in 80 turns. Our longest stretch without a software failure was seven laps.

We present a detailed analysis of these data in the following subsections. First we investigate lane detection accuracy and then combine this with navigation information to evaluate the system as a whole.

4.1 Perception

The task of our lane detection module is to determine the position of the center of the road, marked by a double yellow line, relative to the robot. This localization compensates for errors in global position estimation. Figure 18 shows output from a typical run. To establish ground-truth position, we instrumented the test track with special features observable in the logged laser data. These features are placed at known locations around roughly half of our test track between the labels (a, b, d, and c) in Fig. 17. By manually checking the chosen

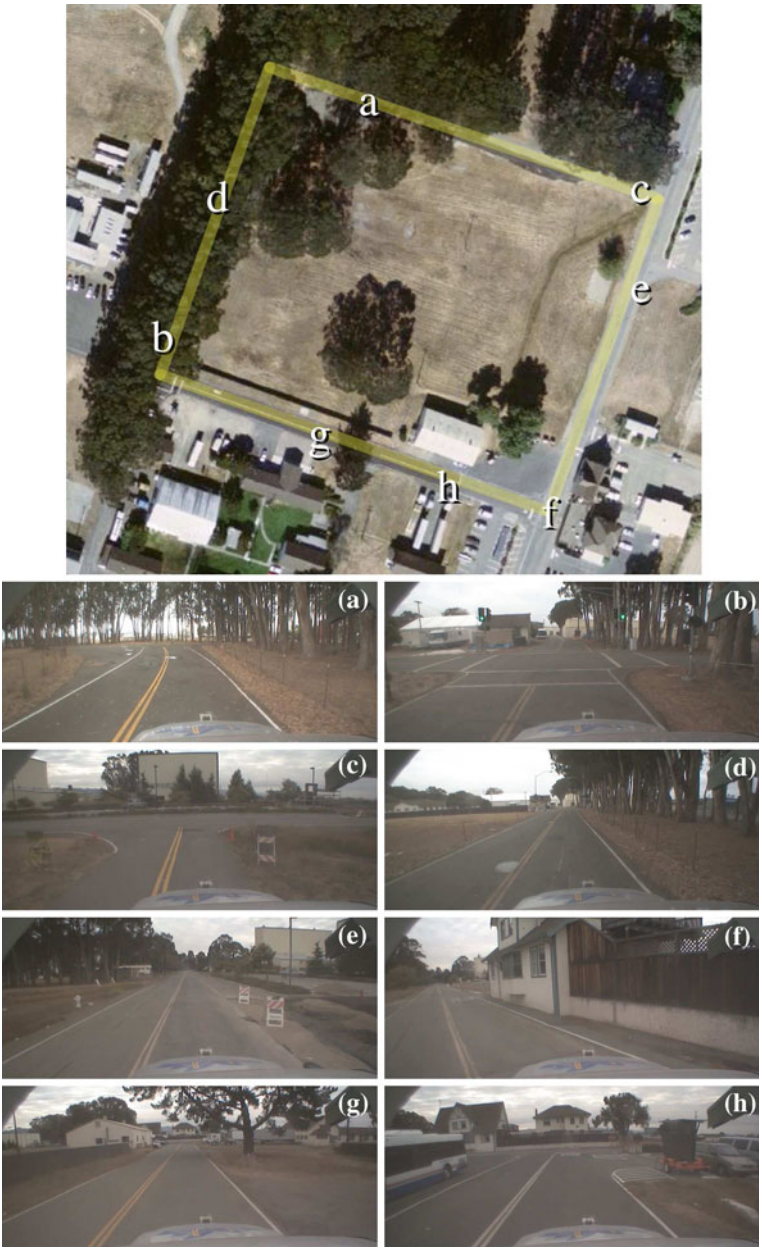


Fig. 17 Aerial view of test track. The *transparent yellow line* represents the 560-meter square loop in which we tested our vehicle. The *letters* indicate correspondence with the lower figures as seen from the vehicle. Note that GPS coverage is minimal along the east and north segments of the loop with areas where no satellites are observed

Table 1 Results summary

	Clockwise	Counter-clockwise	Total
Laps	10	10	20
Elapsed time (min)	51	53	104
Distance traveled (km)	5.6	5.6	11.2
Most consecutive laps without failure	7	4	7
Max speed (kph)	22	22	22
Failures			
Power steering	1	2	3
Bad turn	0	1	1
False obstacle	2	6	8
Planning	1	0	1
Total	4	9	13

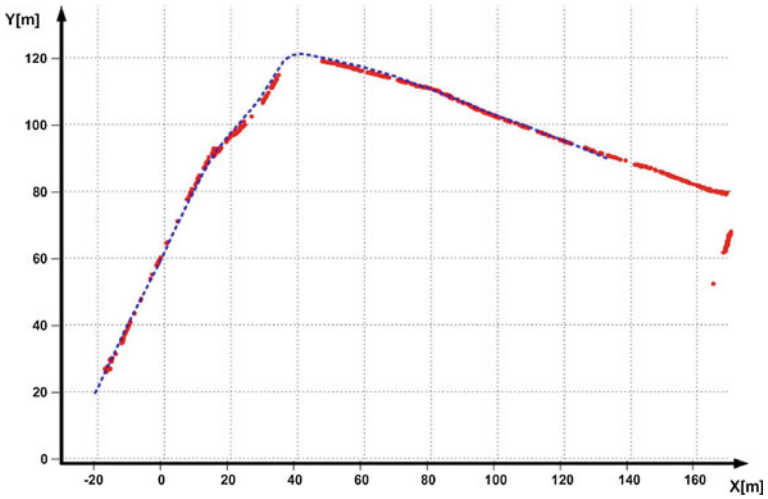


Fig. 18 Comparison between the actual center line (*blue dashed line*) and the lane extracted from vision (*red lines*)

center marking against stored camera images, we found only 0.5% false positives. Mean frames between false negatives was approximately ten. Our system is tolerant to a reasonable number of false negatives since we simply reuse the most recent measurement. As long as we sense a lane relatively soon after leaving an intersection, false negatives are not a problem. Since the number of false positives was low, we take this as evidence that our lane detection module works exceptionally well.

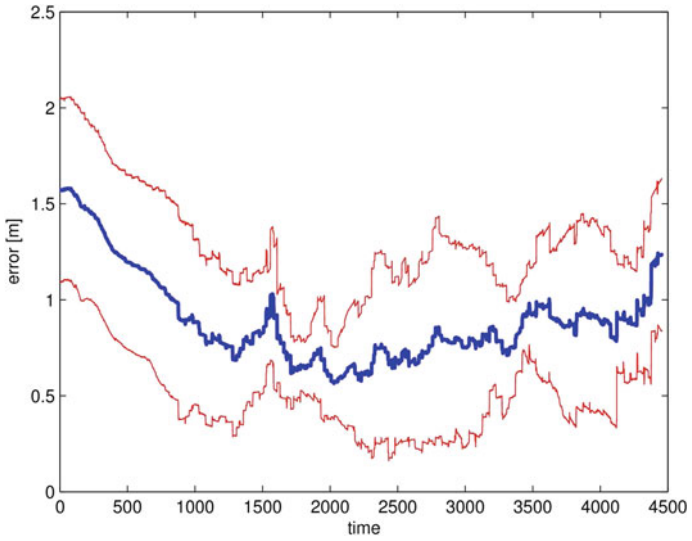


Fig. 19 Global navigation solution error (*blue line*) and one standard deviation (*red lines*) without visual lane correction

4.2 Motion Planning with Lane Detection

The first challenge in analyzing full system performance is to quantify the magnitude of navigation error. From ground truth data, we can compare the actual position of the car to the estimated global position received from GPS. Figure 19 shows that our estimated position changes rapidly and can be up to 1.8 m away from the true position. This error is severe. Relying on GPS alone results in frequently driving completely off the road. Without vision-based lane detection for correction of the navigation solution, we rarely complete one full loop without an unrecoverable failure.

This large absolute error in position estimate can lead to two types of problems: (1) lateral error that leads us to drive out of our lane, and (2) longitudinal error that leads to difficulty in making turns at intersections. Type 2 errors are uncorrected in our current system and we must rely on obstacle avoidance to make turns safely. Type 1 errors are addressed by our lane detection system.

Having established the accuracy of our lane detection system, a final measure of system performance is possible simply by analyzing how well the car tracks the adjusted lane line provided to the path planner. Results for five loops are shown in Fig. 20.

Here we see that the car stayed on its target line with a mean error of 3 cm and a standard deviation of 66 cm. This result is striking when compared to the GPS/IMU trajectories without visual lane correction shown in Fig. 21 with errors

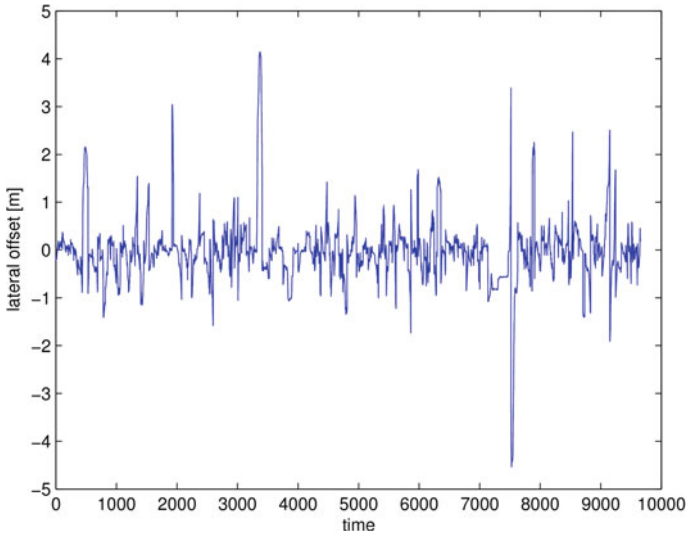


Fig. 20 Error between desired and actual position over five loops

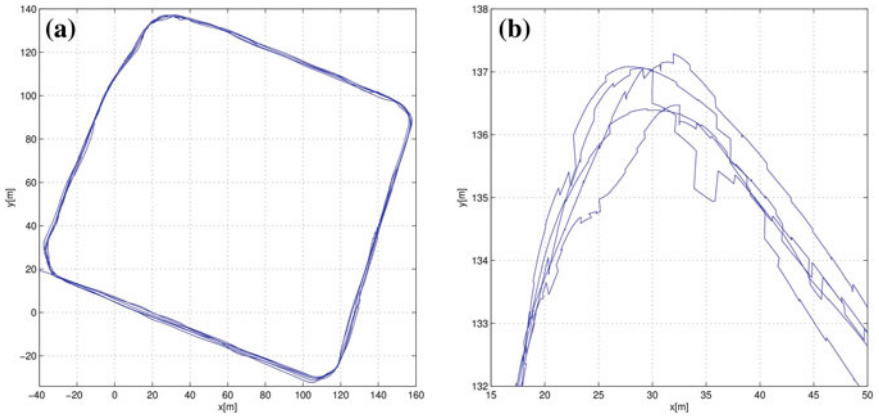


Fig. 21 **a** Overlay of GPS/IMU trajectories for five loops. **b** Magnified view of the top corner in (**a**)

of up to 2 m in the actual navigation solution. The larger lane tracking errors can arise from three possible sources: (1) obstacle avoidance, (2) hardware failure, and (3) entering/leaving intersections.

5 Discussion and Ongoing Work

Of the failures reported in our results, the majority were due to false obstacles and hardware failures in the power steering system. These are serious problems that require manual intervention for recovery. However, we do not feel these are fundamental flaws and both are currently being investigated. We expect that our vehicle will then exhibit reliably good performance on novel stretches of road using lane detection for lateral correction.

This lateral correction does not address problems at intersections, however, since it does not compensate for longitudinal error in position estimates. It appears necessary to integrate information from lasers to give a better local estimate of the geometry of an intersection. This information would help us to make smoother turns.

We have continued work in the model-based expression of our overall component integration strategy [20], which permits us to synthesize simulation experiments from a central location, as well as archive experiments based on regression test criteria (e.g., bugs fixed). Additional work in developing event-based, time-triggered components to standardize executions across platforms and network architectures is paired with this model-based approach, enabling future work in autogeneration of buffer components and execution triggers based on a specific distribution topology [22].

The RRT-based approach is a powerful method for finding paths among complex obstacles. One challenge we encountered was in designing a cost function that favored smooth paths. Paths generated by the planner are composed of randomly-chosen control actions. Thus, there is no explicit smoothness guaranteed. We addressed this by tuning our cost function, however, in hindsight it would have been better to apply aggressive path smoothing as a post-processing step.

Although we have shown robust navigation runs, we have yet to fully validate the intelligent high-level planning capabilities of our system in handling four-way intersections and operating among traffic in general. We did show initial positive results with queuing behavior and overtaking parked cars, and will focus on empirical evaluation in this area in future research.

Robust operation is a primary aim for future work and we believe that reliance on GPS is a primary cause of failure due to signal degradation in proximity to infrastructure and natural features such as buildings and trees. This was continually encountered on our test track as demonstrated in the results section. It is anticipated that both passive and active sensing will ultimately be needed to provide the required fidelity and reliability for autonomous platforms to operate in civilian areas. Our research now focusses on the fusion of traditional laser range information with stereo vision. Vision provides the ability to capture a large field of view in one snapshot with semantic, classification, and recognition hints about each feature from its properties such as color and texture. Stereo vision enables local and global 3-D reconstructions of the environment allowing a lesser degree of dependence on GPS. In combination with laser range finders, we expect to

develop high-integrity local and global maps for navigation resulting in robust autonomous urban driving.

Acknowledgments Thanks to everyone from UC Berkeley and the Sydney team who contributed to the project.This work was supported in part by Rio Tinto, Komatsu, Toyota, Chess at Berkeley, ZeroC, and Advantech. Additional inkind support was provided in the form of discounts on equipment from the following manufacturers: SICK, NovAtel, and Honeywell.The University of Sydney and the University of Technology Sydney are supported by the ARC Centre of excellence program, funded by the Australian Research Council (ARC) and the New South Wales State Government. NICTA is funded by the Australian Government as represented by the Department of broadband, communications and the digital economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Albus, J.: 4D/RCS: a reference model architecture for intelligent unmanned ground vehicles. In: SPIE 16th annual international symposium on aerospace/defense sensing, Simul. Controls (2002)
2. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artif. Intell. **72**(1–2), 81–138 (1995)
3. Bertozzi, M., Broggi, A.: Gold: a parallel real-time stereo vision system for generic obstacle and lane detection. IEEE Trans. Image Process. **7**, 62–81 (1998)
4. Brogdon, J. et al.: Austin robot technology, inc. technical paper. Technical report, http://www.darpa.mil/grandchallenge05/TechPapers/Austin_Robot_Technology.pdf, DARPA grand challenge (2005)
5. Brooks, A., Kaupp, T., Makarenko, A.: Randomised MPC-based motion-planning for mobile robot obstacle avoidance. In: Proceedings of IEEE ICRA, pp. 3962–3967 (2009)
6. Bruce, J., Veloso, M.: Real-time randomized path planning for robot navigation. In: Proceedings of IEEE/RSJ IROS (2002)
7. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of robot motion: theory, algorithms, and implementations. MIT Press, Cambridge (2005)
8. DARPA: E-stop guidelines. Technial report, http://www.darpa.mil/grandchallenge/docs/E-Stop_Guidelines_042307.pdf, DARPA grand challenge (2007)
9. Donald, B., Xavier, P., Canny, J., Reif, J.: Kinodynamic motion planning. J. ACM **40**(5), 1048–1066 (1993)
10. Duda, R., Hart, P.: Pattern Classification and Scene Analysis. Wiley, New Jersey (1973)
11. Elfes, A.: Occupancy grids: a stochastic spatial representation for active robot perception. In: Proceedings of UAI (1989)
12. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. SSC **4**(2), 100–107 (1968)
13. Iagnemma, K., Buehler, M.: Special issue on the DARPA grand challenge, part 1. J. Field Robot. **23**, 461–652 (2006)
14. Iagnemma, K., Buehler, M.: Special issue on the DARPA grand challenge part 2. J. Field Robot. **23**, 655–835 (2006)
15. Ladd, A., Kavraki, L.: Fast tree-based exploration of state space for robots with dynamics. Algorithmic Found. Robot. VI, Springer STAR. **17**, 297 (2005)
16. LaValle, S.: Planning algorithms. Cambridge University Press, Cambridge (2006)
17. LaValle, S., Kuffner, J.: Randomized kinodynamic planning. Int. J. Robot. Res. **20**(5), 378–400 (2001)

18. Rouff, C. (ed.): Second Special Section on the DARPA urban grand challenge, vol. **4**(12). American Institute of Aeronautics and Astronautics, Cambridge (2007)
19. Rouff C. (ed.): Second special section on the DARPA urban grand challenge, vol. **5**(12). American Institute of Aeronautics and Astronautics (2008)
20. Schuster, A., Sprinkle, J.: Synthesizing executable simulations from structural models of component-based systems, vol. **21**(10). Electronic Communications of the European Association of Software Science and Technology (EASST) (2009)
21. Shan, Y., Yang F., Wang, R.: Color space selection for moving shadow elimination. In: Proceedings of ICIG, pp. 496–501 (2007)
22. Sprinkle, J., Eames, B.: Time-triggered buffers for event-based middleware systems. Innovations in Systems and Software Engineering. **7**(1), 9–22 (2011)
23. Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A., Mahoney, P.: Stanley: the robot that won the DARPA Grand Challenge. J. Field Robot. **23**, 661–692 (2006)

Appendix A
Route Network Definition Files

The Route Network Definition Files (RNDF) gave the structure of the roads and open areas that a vehicle was allowed to run on. The RNDF did got give a particular course through the network of roads and open areas, that was contained in the Mission Data File (MDF) which is described in Appendix B. The RNDFs were used for the site visits, the National Qualification Event (NQE) and the final race. For site visits, participants had to come up with their own RNDFs since each site was different. DARPA supplied the RNDF for the NQE and the final event.

The course was described in the file as being made up of either segments or zones (Fig. A.1). The segments were road segments and were made up of lanes, which in turn were made up of a series of waypoints. The zones were made up of a collection of spots and a perimeter. The perimeter is defined as a polygonal boundary that was defined by the perimeter points. The spots were for parking and were defined by two waypoints. A vehicle would move into a parking spot by first going over the first waypoint and then pull over the second waypoint, which was also a checkpoint. A spot was exited by backing up and passing over the first waypoint for the spot again. Vehicles entered zones through the exit point of a segment and exited the zone into the starting point of another segment that was adjacent to the zone.

Figure A.2 gives a graphical view of the components of a road segment. Each segment was given a unique identifier (a number). Each segment consisted of a series of lanes and each lane of a series of waypoints. Waypoints were defined by the triple (segment, lane, waypoint). Each lane was also given an optional width and the segments optional speed limits. The connections between lanes are defined using exit and entry waypoints. A vehicle then can move from the exit waypoint of one lane to the entry waypoint of a neighboring lane. Entry and exit waypoints can occur at the beginning, end, or in the middle of a lane. An example of an exit waypoint in the middle of a lane would be at an intersection. The intersection would have exit waypoints for each lane with associated entry waypoints for lanes the vehicle can turn into. Stop signs in are defined in the RNDF as part of a waypoint, as are checkpoints.

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3, © Springer-Verlag London Limited 2012

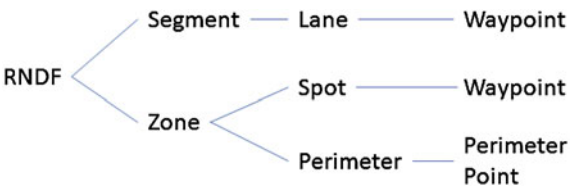


Fig. A.1 Route Network Definition File (RNDF) structure

Figure A.3 shows a part of the RNDF used for the course in the final event of the Urban Challenge. The first part of the file has comments (between the "/*" and "*/" are comments). Following the comments is the header information for the file which includes the name of the file, the number of road segments, the number of zones, a version number, and a creation date.

Below the preamble information are the segment definitions. The first segment of the 60 segments is shown. It starts with the segment number followed by the number of lanes in the segment and the segment name. Following the segment name is the definition for the first (and only) lane in this segment. This lane has seven waypoints and has a width of 12 feet. The exits give the exit waypoint number from the current lane to the waypoint number of the entry point of another lane. For this lane there are exit waypoints at the 4th through 7th waypoints. The first exit waypoint for the lane has an entry point of segment 61, lane 0, and waypoint number 8.

Following the exit points are the waypoint definitions. There are seven waypoints for this lane. Each waypoint has a number consisting of the segment number, the lane number, and a number of the waypoint in the lane, followed by the latitude and longitude of the waypoint. The first waypoint shown, numbered 1.1.1 for segment 1, lane 1, is at latitude 34.587489 and longitude −117.367106 and are given in decimal degrees.

Figure A.4 shows an example of a zone definition. This zone is the 8th zone and is numbered 68 (numbering continues after the segments) of the file. There is one

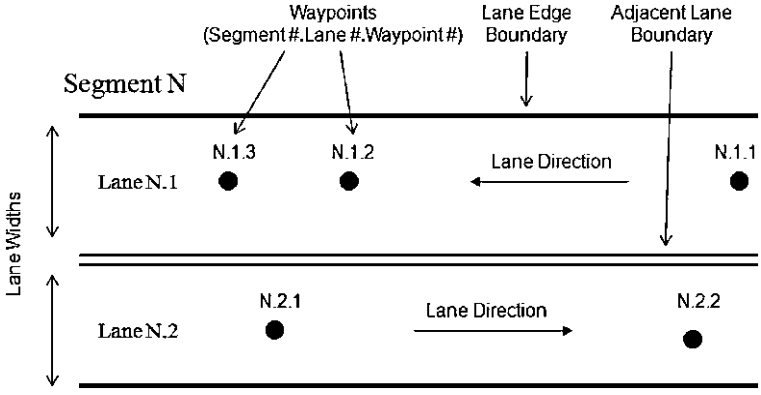


Fig. A.2 Graphical depiction of the parts of a road segment

```
/*          Urban Challenge Final Event RNDF          */
/*                  November 2, 2007                  */
/* ============================ */
/*  Changes from previous version:  */
/*  Added three way stop at intersection of Texas and Idaho  */
/*  Corrected segment 30 (West side of circle) numbering   */
/* ============================ */
RNDF_name uce_rndf_1
num_segments 60
num_zones 8
format_version 1.0
creation_date 3-Nov-07
/* ============================ */
segment 1
num_lanes 1
segment_name red_zone_ElotAccessRd
lane 1.1
num_waypoints 7
lane_width 12
exit 1.1.4 61.0.8 /* into Red Zone Center */
exit 1.1.5 61.0.7 /* into Red Zone Center */
exit 1.1.6 62.0.7 /* into Red Zone East */
exit 1.1.7 62.0.6 /* into Red Zone East */
1.1.1 34.587489 -117.367106
1.1.2 34.587120 -117.367107
1.1.3 34.587069 -117.367021
1.1.4 34.587069 -117.366937
1.1.5 34.587071 -117.366530
1.1.6 34.587070 -117.366017
1.1.7 34.587045 -117.365589
end_lane
end_segment
```

Fig. A.3 File preamble and segment definition in a Route Network Definition File (RNDF)

parking spot in the zone and the zone has a name of "Start_zone". After the zone name is the definition of the perimeter in the zone. The perimeter ID is the same as the zone ID with a ".0" after it. The perimeter ID is an artifice and is there so there is a uniform naming convention for the different elements of the RNDF. After the perimeter ID is the number of points in the perimeter definition. In this case there are 30 points. In this case the last perimeter point is defined as an exit point (68.0.30) and the corresponding entry point is segment 2, lane 1, and waypoint 1.

The parking spot is defined at the bottom of the zone definition. It has a number that consists of the zone number (68) followed by the number of the spot in the zone (1). The checkpoint shows that the spot is a named waypoint in the zone and has the same format as a lane waypoint, zone number, lane number, and waypoint number. Each waypoint is also defined by its latitude and longitude. The second waypoint of a spot is always a checkpoint. The vehicle must pull over the first waypoint and to the checkpoint.

```
/* ============================ */
zone 68
num_spots 1
zone_name Start_zone
perimeter 68.0
num_perimeterpoints 30
exit 68.0.30 2.1.1 /* leave start zone onto Challenge Lane */
68.0.1 34.584775 -117.369108
68.0.2 34.584775 -117.368890
68.0.3 34.584728 -117.368890
68.0.4 34.584689 -117.368890
68.0.5 34.584649 -117.368889
68.0.6 34.584610 -117.368889
68.0.7 34.584570 -117.368888
68.0.8 34.584530 -117.368887
68.0.9 34.584491 -117.368887
68.0.10 34.584451 -117.368886
68.0.11 34.584412 -117.368886
68.0.12 34.584372 -117.368885
68.0.13 34.584333 -117.368885
68.0.14 34.584293 -117.368884
68.0.15 34.584253 -117.368883
68.0.16 34.584214 -117.368883
68.0.17 34.584174 -117.368882
68.0.18 34.584135 -117.368882
68.0.19 34.584095 -117.368881
68.0.20 34.584055 -117.368880
68.0.21 34.584016 -117.368880
68.0.22 34.583976 -117.368879
68.0.23 34.583937 -117.368879
68.0.24 34.583897 -117.368878
68.0.25 34.583858 -117.368878
68.0.26 34.583818 -117.368877
68.0.27 34.583680 -117.368876
68.0.28 34.583680 -117.368956
68.0.29 34.583680 -117.369108
68.0.30 34.583765 -117.369108
end_perimeter
spot 68.1
spot_width 12
checkpoint 68.1.2 76
68.1.1 34.583721 -117.368955 /* Finish Spot Waypoint */
\index{waypoint}\index{navigation!checkpoint}
68.1.2 34.583738 -117.368955 /* Finish Spot Checkpoint */
end_spot
end_zone
end_file
```

Fig. A.4 Zone definition in the Route Network Definition File (RNDF)

Appendix B
Mission Data File

The Mission Data File (MDF) contained the definition of a particular mission that was to be performed by a vehicle. The MDF was given out at least 5 min before the start of the event and then at least 5 min before the start of the second mission. It contained a sequence of checkpoints that a vehicle had to visit. The checkpoints were defined in a corresponding RNDF (see Appendix A). Checkpoints had to be visited by traveling in the defined lane and in the correct direction, and a checkpoint could be listed multiple times in the MDF. Not all checkpoints defined in the RNDF have to be used in an MDF. The MDF also contained maximum and minimum speeds for segments and zones that were traveled. The last checkpoint in an MDF was the finish line.

Figure B.5 shows the structure of an MDF. It contains a series of checkpoint IDs that the vehicle is to cross and a series of speed limits. The checkpoints are defined in an RNDF, which is named in the header section of the file. The checkpoint IDs are defined in the named RNDF. The speed limits are for segments and zones that are named in the RNDF and are for all lanes of a segment and areas of a zone. A minimum and a maximum value of a speed limit (in miles per hour) is given for the segment/zone. If a zero is used, then the speed limit is undefined. If the minimum and maximum speeds equal each other, then that speed must be maintained for the segment or zone.

Figure B.6 shows a sample MDF from the Urban Challenge final event. The first part of the file is the header which gives the name of the MDF, the RNDF that

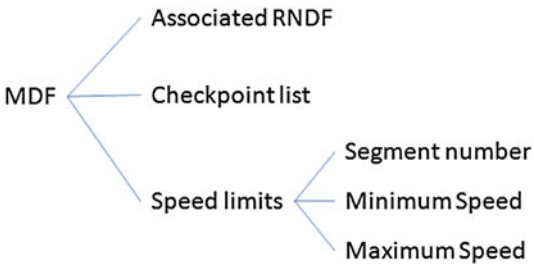


Fig. B.5 Mission Data File (MDF) structure

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3, © Springer-Verlag London Limited 2012

```
\index{Mission Data File}\index{navigation!checkpoint}
/*        Urban Challenge Final Event MDF Sample       */
/* ================================================== */
/* This MDF represents a typical mission from the UCE, */
/*  starting in the start zone, visiting checkpoints   */
/*  throughout the course, and finishing on Finish Rd  */
/* ================================================== */
MDF_name uce_mdf_sample
RNDF uce_rndf_1.txt
format_version 1.0
creation_date 15-Nov-07
checkpoints
num_checkpoints 38
47 2 34 82 83 26 5 1 2 6 153 26 5 1 2 26 18 25
19 5 1 4 8 27 21 3 1 4 14 1 4 18 20 25 23 7 3 76
end_checkpoints
speed_limits
num_speed_limits 68
1 5 10 /* red_zone_ElotAccessRd */
2 5 10 /* Challenge_Ln */
3 10 15 /* Main_Cir_eastside */
4 5 5 /* Pit_Ln */
5 5 5 /* Victory_Ln */
6 10 20 /* George_Blvd */
7 10 20 /* Sabre_Blvd */
8 5 10 /* Red_zone_access_Rd */
9 10 20 /* North_Nevada */
10 10 20 /* South_Nevada */
11 10 20 /* Washington_St */
12 10 20 /* Carolina_Ave */
13 10 20 /* Montana_St */
14 10 20 /* Utah_St */
15 10 20 /* Oregon_St */
16 10 20 /* Michigan_Ln */
17 10 20 /* Ohio_Ln */
18 10 20 /* Kentucky_Ln */
19 10 20 /* Indiana_Ln */
20 10 20 /* Tennessee_Ln */

[speed limits 21 - 59 deleted] \index{speed!limit}

60 5 10 /* Red_zone_Wlot_Access_Rd */
61 5 10 /* Red_zone_Center_Lot */
62 5 10 /* Red_zone_East_Lot */
63 5 10 /* Red_zone_West_Lot */
64 5 10 /* White_zone_north */
65 5 10 /* White_zone_south */
66 5 10 /* Blue_zone_north */
67 5 10 /* Blue_zone_south */
68 5 10 /* Start_zone */
end_speed_limits
end_file
```

Fig. B.6 File preamble and segment definition in a Mission Data File (MDF)

this MDF applies to, the version and the date it was created. After the header the checkpoints are defined. In this case there are 38 checkpoints for this mission. After the checkpoints, 68 speed limits are defined. The speed limits are listed by the segment number. Segment number 1 has a minimum speed limit of 5 mph and a maximum of 10 mph. The name of the street or zone is listed in the comments. The middle segment speed limits (21–59) have been removed for brevity.

Index

A
Ackerman steering, 215
Action selection, 98, 107
Advantech, 276, 300
Agents, 44, 94
AIAA Journal of Aerospace Computing, Information and Communication, 15
Air Force Research Laboratory, 89
Algorithms
  bresenham, 223
  dijkstra, 55, 98
  expectation maximization, 80
  greedy, 52
  hough transform, 79
  hungarian, 219
  nearest neighbor, 160, 171, 190, 219
  newton's root approximation, 159, 171
  rapidly-exploring random trees, 292
  search, 97–98
    A*, 71, 98, 106, 112, 142, 286
    model-predictive, 128
    trees, 294
Arbitration, 22–25, 77, 79, 89, 107, 109, 171, 190, 219
Architecture, 32, 38, 50, 55, 68–70, 80, 88, 94–96, 99–103, 105, 107, 111, 113–117, 119, 123, 126, 128–210, 215, 288
  3T, 99
  autonomous robot architecture, 101
  behaviorial, 50, 99
  deliberative, 15, 100–101, 111–113, 128
  flexibility, 121
  hybrid, 99, 100–101, 128
  reactive, 50, 94, 98–99, 100–102, 107, 128
  repeatability, 101
  saphira, 100
  scalability, 94, 101
  subsumption, 24, 100
  system, 94, 260, 274
  task control architecture, 100
Artificial intelligence, 24, 128, 259
  behavioral, 110
  biological, 99, 107, 128
  contextual, 102, 107, 128
  cooperative, 107
Austin Robot Technology Team, 277
Australia National Information and Communications Technology, 299
Australian Research Council, 299

B
Bayesian probability, 223
Behavior, 73, 80–81, 83–84, 86, 88, 94, 98–99, 102, 107-112, 120, 128, 215, 234, 238
  anomalies
  collaboration, 108
  complex, 108, 111
  decision, 109, 213
  deliberative, 98
  driving, 102, 107, 111, 113, 115, 119–120, 122, 124, 126–127
  emergent, 98, 102, 108, 128
  learning, 126
  modes, 70, 89
  profile, 107, 111, 113, 124
  reactive, 98–99, 105, 107
  reflexive, 120
  sequencing, 100
  tactical, 50
  temporal, 102
Ben Franklin Racing Team, 13

C. Rouff and M. Hinchey (eds.), *Experience from the DARPA Urban Challenge*,
DOI: 10.1007/978-0-85729-772-3, © Springer-Verlag London Limited 2012

B (*cont.*)
Berkeley DB, 35, 264
BERRA, 23
BitCtrl systems, 279
Bluestorm, 281–282
Boost spirit, 263
Boss, 7, 14, 20
Brain, 99
Broker, 81–83
Btrfs, 254

C
C++, 36, 38, 247, 252, 254–255
C&C C-Nav, 197, 199
CajunBot-II, 196, 198
California
  barstow, 5
  hesperia, 260
  speedway, 5
  traffic laws, 6, 10–11, 55, 105, 120
  victorville, 6, 11, 45, 249–250
Camera, 20, 27, 29, 44, 48–50, 53, 78–79, 134, 197, 213, 227–228, 237–239, 265–266, 274, 280–282, 285, 289, 295–296
  color, 75, 97, 222, 227, 266, 289
  hitachi, 280–281
  matrix vision, 75
  monovision, 52
  prosilica, 47
CARMEN, 22–23
Carnegie Mellon University, 5–6, 14
Caroline, 210, 245
CarOLO Project, 244, 249
Caterpillar, Inc., 129–130
Center for Hybrid and Embedded Software Systems (CHESS), Berkeley, 299
Chevrolet, Tahoe, 13–14
CLARAty, 22
Cognition, 94
Command fusion, 107–110, 126, 128
Commercial off-the-shelf, *see* COTS, 31
Communication Database with Geometric Reasoning (CODGER), 24
Communications, 26, 35, 39, 101, 117, 261
  latency, 124–126
  protocol, 129
  synchronous, 25–26
Compass, 188
Computer hardware, 44–45, 49, 96, 116, 197, 280
Concurrency, 107–108, 123, 127–128
Continental, 39
Control, 87, 98–99, 101, 107, 111, 115, 134, 229, 233–234, 282, 285
  engineering, 210–211
  lateral, 234–235
  longitudinal, 233–234
  model predictive control, 85
  motion, 99, 101, 211, 237
  policies, 110, 126
Cook, 254
CORBA, 23, 33
  Ruby, 36
Cornell University, 13
COTS, 30, 44, 97, 103
  integration, 94
CruiseControl, 267
CxxTest, 254–255, 265, 267

D
DARPA, 4, 7, 15, 39, 44, 52
  crusher, 44
  grand challenge, 4, 7, 21, 55, 67–68, 71, 78, 94, 134, 210, 243, 249, 274, 276–277, 282–283, 289
  learning applied to ground robots, 44
  unmanned ground combat vehicle perceptor integration, 44
Data
  acquisition, 31, 219
  association, 219
  diagnostics, 27, 35
  error, 268
  fusion, 77–79, 94, 96, 100, 129, 213–214, 220, 223, 229, 238–239, 274, 281, 299
  integration, 228
  logging, 21–22, 34, 35, 117
  playback, 21, 30, 117, 198
  sensor, 27, 102, 213, 218, 222
  visualization, 118, 198
Decision support, 82, 95, 102–103, 105, 118–119, 128, 260
Defense Advanced Research Projects Agency, *see* DARPA, 4
Dell, 199
Dempster's rule of combination, 229
Dempster-Shafer distribution, 222–223, 226–227, 238
Distributed computing, 23, 31, 96
Domain
  knowledge, 112
  specific language, 262, 267

Drivability, 213, 229, 237
analysis, 221, 223, 225, 227, 228
model, 221

E
E-Stop, 9, 278
Hetronic ERGO V4, 279
Eclipse, 252, 254, 262
Eigenvalues, 225
Electrical engineering, 210
Electronic
control unit, 87, 122
mobility control, 45, 199
Emergency-Stop, *see* E-Stop
environment, 107, 112
dynamic, 44, 81, 94, 128, 134–135, 137–138, 142
height, 223, 225
model, 221
unpredictable, 128
unstructured, 127
Errors, 115, 127–128
component-level, 127
detection, 115, 246
hardware, 119, 122, 294, 298
recovery, 21, 94, 126–127
Ethology, 98, 108
Euclidian point distance, 219

F
Fedora core, 199
Finite state machine, 51, 108
hierarchical, 108, 126
behavioral, 108
hybrid, 55, 57, 65
FireWire, 282
Ford, 130
escape hybrid, 96
F-250, 13
Free navigation zone, 9–11
Fresnel, 164
integrals, 164
sine and cosine, 162
FSVS, 253–254

G
Gcov, 255
General electric aviation, 89
north finding module, 74
General motors company, 39
George air force base, 6, 11, 249
Georgia
public safety training center, 60–61
tech, 44, 65
Germany, 12–13, 210, 250
Global positioning satellite, *see* GPS, 4
GNU, 96, 256
Google, 33
GPS, 4, 10, 44, 48, 50, 52–53, 58, 60, 63–64, 68, 74–75, 97, 103, 188, 198–200, 202, 206, 214–215, 274, 281–282, 285–286, 297, 299
correction, 52, 97, 199
OmniSTAR HP, 282
WAAS, 49
error, 49, 63, 104, 199, 274, 297
garmin, 74
NovAtel, 48, 282
outage, 10, 75, 104, 274
shifts, 206

H
Health monitoring, 34, 115–116, 123
Hesperia framework, 264
Hetronic ERGO V4, 279
Hewlett-Packard, 96
HMMWV, 5
Honeywell, 13, 49, 300
HG-1700, 282
Humanoid robot, 129

I
IAV Automotive, 269
IBEO Automotive Systems, 96, 116, 122
Image compression, 290
Inertial measurement unit, 274
Inertial navigation system, 97, 103
Information science, 210
Intel, 96, 280
Interprocess Communications Toolkit (IPT), 23
Inverse perspective mapping, 289
IPG automotive simulation, 268
Iteris lane departure warning system, 199

J
Jacobian, 221
JAUS, 68, 81–82, 94, 116–117, 123, 128
autoconfiguration, 116
messaging, 116–117, 123, 128
SAE AS-4, 116
simulation, 117
toolkit, 117

J (*cont.*)
Java, 38, 263
Jeep wrangler rubicon, 198
Joint architecture for unmanned systems, *see* JAUS
Journal of field robotics, 103
Junior, 7, 14

K
Kalman filter, 74, 103, 220
Knight rider, 137
Komatsu, 300

L
LADAR, 68–69, 78–80, 88
  Doppler shift, 218
  ground reflections, 220
  SICK, 75
Land rover, 13
Lane, 8–9, 11, 20, 28, 37, 50, 52, 54, 57–58, 60–61, 69–72, 74, 77, 79, 81–82, 84–85, 88, 101, 103–106, 110–113, 115, 119, 122, 179–180, 183, 186, 190, 198, 201, 207, 231, 250, 263, 274–275, 282, 286–288, 290, 292, 294, 296–297, 301, 307
  change, 54, 61, 82, 97, 106, 121, 174, 180–181, 186
  correction, 79–80, 88, 297, 299
  detection, 52, 68, 79, 128, 282, 288, 294, 296, 299
    vision-based, 297
  following, 47, 84–85, 198
  markings, 9–10, 57, 227, 231, 262, 274, 288, 290
  position, 104, 116, 124, 201
  tracking, 53, 102, 207
Law of sines, 177, 179
Learning, 126, 259
Least-squares fit, 217
Lehigh university, 13
LIDAR, 44, 46, 48, 50, 52, 58, 63–64, 96–97, 122, 199, 212, 217, 223, 237
  IBEO, 96–97, 105, 122, 199
  limitations, 105
  riegl, 47–48
  SICK, 46–47, 97, 199, 213, 223, 300
Linux, 96, 116, 197, 280
  fedora, 96
  kernel, 280
  kubuntu, 280, 284
  virtual machine, 116
Localization, 23, 74–75, 77, 79–80, 103–104, 116–117, 124, 127, 198–199, 200, 210, 213, 274, 289, 294
  global positioning, 50, 71, 285
Lockheed Martin, 13
Louisiana Governor's Information Technology Initiative, 207

M
Make, 254
Maneuvers, 172, 179–180, 186–187, 229, 235
  evasive, 237
  feasible, 183
  generic-turn, 173, 177
  illegal, 184
  intersection, 70, 84, 302
  lane change, 186–187
  left-turn, 183–184
  merging, 7, 119
  N-point turn, 84
  parking, 113, 119, 249
  passing, 5–8, 10, 20, 83, 110, 236
  reverse, 70, 83, 86, 122, 184, 230
  right-turn, 183, 186
  stop, 8, 10, 20, 34, 53, 56, 58, 61, 63, 65, 73, 84, 86, 224, 237, 249, 276, 279, 292
  u-turn, 8, 11, 57, 60, 63, 121, 159, 173, 179, 181, 183, 186, 190, 215, 230–232, 237, 294
Map, 104, 112, 142, 214, 221, 228, 249, 283, 286, 299
  grid, 69, 77–78, 85, 222–224, 226, 287
  making, 99, 101, 129
  terrain, 128, 274, 282
  traversability, 70, 77–78, 80, 85, 86, 287, 290
Massachusetts Institute of Technology, *see* MIT, 69
MATLAB/Simulink, 252, 268
MDF, *see* Mission data file
Messaging, 23, 113, 116, 120–121, 123–124, 126–128
  neutral messaging language, 23
Microsoft
  powerpoint, 254
  robotic studio, 22
  windows server, 116
  windows XP, 96

Miro, 22–23, 33
Mission
goals, 73
objectives, 211
Mission data file, 9, 54, 69, 95, 134, 302
road segment, 307, 309
speed limit, 307, 309
zone, 307
MIT, 13, 237
Modeling, 282–283
3 dimensional, 264, 268
bicycle, 235
environment, 223, 283
local world, 69, 72–74, 79, 85
mathematical, 261
movement, 216
sensor, 228–229
terrain, 274
vehicle, 124, 137–138, 197, 200, 203, 263
world, 24, 94, 102–103, 106, 211, 213, 222–223, 259
ModUtils, 22, 25
Monitoring, 102
health, 95, 101, 120, 122
performance, 99, 107
progress, 122
MontiCore, 262
Motion, 99, 102, 104, 107, 113–114, 118, 211, 217, 234, 237
Movies, 30
MOXA, 116
MPEG, 31
Multicast, 33, 265

N

NASA, 22
National
ICT Australia, 274
Instruments CompactRIO, 96
NAVCOM Starfire Differential GPS, 197
Navigation, 48, 57, 78, 100, 104, 128, 206, 210, 268, 280, 282–283, 285, 290, 294, 299
behavior-based, 101
checkpoint, 8–9, 37, 73, 95, 105, 121, 236, 285, 302, 304, 307, 309
collision, 63–64, 126, 149
avoidance, 134, 135–136, 143, 145
DAMN, 214
dead reckoning, 274
direction, 111
distance estimation, 73, 239
errors, 235, 296
exit points, 73
failure, 294, 299
free zone, 9, 70–72, 83, 85, 88, 104, 106, 111, 119, 122, 236, 238–239, 301
heading, 85, 113, 205
inertial, 48, 67, 74–75, 103, 213, 279, 281, 284
intersection, 6–8, 10, 20, 26, 28, 37, 47, 50, 52–53, 57, 60–61, 63, 67, 69–70, 84–86, 88, 104, 106, 109, 111, 118–120, 123, 126, 179, 205, 211, 215, 236–237, 249–250, 263, 275, 286, 294, 296–297, 299
merge, 120, 126, 249–250
off road, 70, 84, 128
parking, 1, 3, 7, 9–11, 19, 37, 57, 60, 64, 70, 83–84, 86, 113, 122, 215, 230–231, 237, 249–250, 282
passing, 7, 63, 83, 85, 110, 179, 249
path, 71, 84–85, 86, 202, 206–207
position estimation, 103, 218
receeding horizon control, 73, 85
right of way, 7, 84
road, 60, 83, 85–86, 88
blocked, 6–7, 10–11, 54, 58, 83, 105, 107, 119, 231, 236, 250, 286
satellite, 213
starting dock, 64
stop sign, 9, 11, 63, 120, 126, 286
traffic, 78, 95, 104, 106, 112, 115, 119–120
circle, 6, 64, 67, 88
traversability, 86, 88, 287
Network, 32, 96, 199, 279–280
ethernet, 45, 48–49, 50, 96, 116, 123, 199, 280
latency, 123, 280
protocol, 33, 129
UDP, 33, 264
wireless, 199
Network data distribution service, 23
Nevada, Primm, 5
New south wales state government, 300
Night rider, 134
North finding module, 74
NovAtel, 300
ProPak, 74, 97, 282

O

Object, 122, 248
acceleration, 221

O (*cont.*)
classification, 52, 73, 97, 104–105, 120, 123–124
contour, 219–220
matching, 219
model, polygonal, 220
Obstacle, 112, 274
avoidance, 60, 84, 101–102, 105, 120, 122, 134–135, 141, 143, 145, 149, 297
collision, 112, 126
detection, 10, 47, 50, 52, 67, 75, 79, 88, 97, 104–105, 120, 124, 127, 139, 200, 212, 236, 282, 285, 287–288, 294, 297
dynamic, 6, 50, 52–54, 64, 69, 73–74, 84, 102–105, 112–113, 116, 120, 122–123, 125, 127, 135, 145, 149, 152, 199, 217, 231, 249, 262, 300
prediction, 104, 120
hard/soft, 135, 142, 144, 147–149, 152, 154
localization, 77, 124
static, 50, 52, 69, 74, 83, 86, 95, 102–104, 112, 116, 120, 123, 125, 152, 199, 231, 236
tracking, 20, 65, 79, 104, 213–215, 218, 220, 237, 285
velocity, 215, 218, 220
Odin, 7, 14, 96
OmniSTAR HP, 97, 282
OpenCV, 264
OpenGL, 28, 265–266
Operating system, 280, 287
Operational Software Components for Advanced Robotics (OSCAR), 23
Orca, 24, 283–284
Orocos, 23
Oshkosh Truck, 5, 13
Oxford technology solutions, 199
RT3102 INS, 199

P
Parallelism, 111
PEAK Systems, 279
Perception, 19, 21–22, 24–25, 35, 44–45, 50–52, 69, 71, 74, 77, 82, 84, 94, 102–103, 105, 111, 113–114, 117, 120, 127–129, 210, 214, 229, 237, 260, 281–282, 285–286
image warping, 289
perspective effect, 289
scalability, 103
terrain, 77
vision, *see* vision
Performance
index, 135
monitoring, 99
Pioneer robot, 100
Planning, 20–22, 24–25, 35, 50–51, 54, 68–71, 73, 80, 82, 84–86, 88, 94, 98–99, 101, 103, 114, 117, 119, 123, 127, 134–136, 138–139, 145, 152, 157, 160, 184, 207, 210–211, 213, 229, 231, 233, 237, 250, 254, 274, 279, 282, 285–286, 288, 291, 293–294, 299
deliberative, 99, 101
dynamic programming, 286
goals, 99, 101, 105, 107, 113, 120, 291
mission, 44, 50, 94, 99, 101, 107, 128
motion, 86–87, 94, 101–102, 104, 107, 111–116, 119–125, 129, 134, 157–160, 167, 173, 179, 184, 186–188, 190–191, 218, 229, 234, 237, 279, 282, 285–286, 291, 297
continuous curvature paths, 157
deliberative, 101, 105, 112
dynamic, 135, 292
testing, 188
optimal, 101, 128
path, 54, 71, 94, 101–102, 104–105, 111–112, 120–121, 128, 134, 139, 158, 160, 173, 184, 186, 188, 198–200, 205–207, 230–231, 282, 285, 291–292, 297, 299
rapidly-exploring random trees (RRT), 292
replanning, 139
sample-based, 274
search trees, 292
short-range, 104
street-level, 285–287
time penalties, 106
town, 285
trajectory, 134, 136–139, 142, 145, 236
velocity, 114, 135, 265, 286, 292–293
weighted cost, 292
Player/Gazebo, 22, 293
Player/Stage, 22
Point cloud, 52, 287
Point reduction algorithm, 189–190
Policy, 34, 109–110
Porsche, Cayenne, 44–45
Position, global, 113
Process

communication, 21, 23, 25, 31–32, 38, 68, 115–116
launching, 33
management, 21
single-threaded, 27
Processing cycle time, 123
Prometheus projects, 210
Proportional-Integral-Derivative, 279
Publish/subscribe, 25, 30–31, 35

Q
QMake, 254
QNX, 277, 279–280, 284, 288
neutrino, 280, 284

R
Race, 4–6
course, 11, 63–64, 126, 250
final event, 10–11, 106
finalists, 4–6, 8–9, 11, 14, 21, 106, 235–236, 249–250, 257, 301–302
national qualification event, 5, 10, 12, 45, 60, 62, 64, 250, 257, 301
prize, 5, 6
rules, 231
RADAR, 20, 44, 47, 49, 50, 52, 64, 198–199, 212, 218, 238
Eaton, 47, 52, 197
Real-time
communications (RTC), 23
control systems (RCS), 23, 81
innovations (RTI), 23
Reasoning, 20–21, 24, 82, 98, 134, 283
Red Team, 5, 21
Requirements, 19–24, 27, 30–31, 38, 45, 89, 142, 148, 179, 244, 246, 248, 252, 254, 257–259, 274
reuse, 38
Resource usage, 24
Richmond Field Station (RFS), 293
Rio Tinto, 299
RNDF, *see* Route Network Definition File
Road, 105
analysis, 223, 227, 237
detection, 52, 104–105, 116, 120, 122, 124, 128, 212
drivability, 124
lane, see lane
segment, 1–3, 7, 9, 50, 53–54, 60, 86, 105, 285
virtual, 128
RoboCups, 243
Robustness, 82, 95, 210
Route network definition file, 1–5, 9, 37, 52, 69, 95, 134, 285
Rsync, 34
RWTH Aachen University, 267

S
Safety, 4, 8, 45, 81, 113–114, 249, 277–278, 283
Sandstorm, 5
Saphira, 23
Science Applications International Corporation (SAIC), 44
Self awareness, 107
Sensor, 46, 94, 96, 99, 102–103, 108, 116–117, 122, 127, 217, 236, 274, 280, 284
acoustic, 134
data, 222, 264, 269
synthetic, 265–266, 268
field of View, 269
integration, 68
latency, 280
model, 226, 228
pushbroom, 47
range, 125
virtual, 94, 103, 107, 109–110, 118, 127, 129, 203, 205–206
SimpleComms, 32, 35, 37
Simulation, 22–23, 27, 29–30, 89, 95, 112, 117, 126, 129, 136, 150, 152, 187, 200, 247, 252, 268–269, 299
hardware-in-the-loop, 89, 118
physics-based, 247, 293
system, 244–245, 248, 250, 252, 254, 257, 259–260, 261–263, 266, 268–269
traffic, 120, 269
Simultaneous localization and mapping, 127
Site visits, 4, 8, 250–251
Situation assessment, 70, 72, 81, 211, 213, 237
Situational awareness, 55, 102, 107
Software
bug tracking, 254
build, 254–255
configuration management, 21, 246, 253–254, 269
automatic, 117
dynamic, 116, 198
continuous integration, 254, 268
diagnostics, 22
failure, 26
flexibility, 25

S (*cont.*)
framework, 22–23, 27, 31, 36, 80, 82, 88, 250, 260–266, 268–269
integration, 94, 167, 244, 246, 256, 265, 268
continuous, 246, 248, 254, 257
interactive, 257
latency, 125–126
livelock, 64
management, 246, 254
non-determinism, 27
quality, 15, 24–25, 33, 244, 246–247, 249, 254–255, 257, 267, 269
race conditions, 244, 259
reuse, 114–115, 117, 250, 265, 283
specification, 257, 260
state, 217, 274
stateless, 26, 35, 38–39
Software engineering, 24, 249, 257
best practices, 248
coding guidelines, 255
collaboration, 246
component-based, 283
correctness, 27
design patterns, 248
design-by-contract, 264
distributed development, 252
error, 247, 256, 259
inheritance, 36, 38, 256
knowledge management, 254
memory checking, 255
object-oriented programming, 99
portability, 23, 44, 99, 101, 126–127, 129
predictability, 111
process, 117, 245–246, 252, 255–257
agile, 244, 248, 252, 257, 269
crystal, 244
deadline-oriented, 244
extreme programming, 244, 248, 252, 254
iterative, 246, 269
pair programming, 254
RUP, 246
scrum, 244, 246, 257
story cards, 248, 252, 254, 257, 262
test first, 244, 247, 254, 257, 259
reliability, 37, 95, 101, 210, 299
repeatability, 27
reuse, 283
robustness, 25
safe modes, 115
scenarios, 257, 260, 262
testing, *see* testing, 188
tools, 244, 254–255
automated, 257
debugging, 39
universal modeling language, 252, 254
virtualized, 260
Southwest Research Institute, 8, 250
Speed, 58, 86, 111, 113, 115, 119–120, 125, 198, 202, 205, 211, 215, 233–235
control, 279, 292
limit, 9, 84, 86, 95, 105–106, 249, 285
limiter, 112–113
markers, 291
maximum, 111, 113, 200, 206, 287
optimization, 230, 233
Stanford, 7
Racing Team, 119
Research Institute, 100
University, 5, 14
Stanley, 5
Statecharts, 252
Steering, 45, 50, 58, 60, 68, 70, 85–87, 115, 137, 139, 142, 157, 159, 167, 169, 184, 198–207, 211–212, 216, 231–232, 234–235, 276–277, 279, 294, 299
actuator, 201, 206
angle, 103, 124, 142
control, 86, 198–199, 201, 203, 207, 279
dampening, 204
error, 204–205, 207, 279
failure, 294, 298
lag compensator, 201, 203
smoothness, 207
stability, 204, 207
target, 203, 205
testing, 206
tracking, 204–205
Sting Racing Team, 44
Subaru Outback Legacy 4WD, 14
Subversion, 253–254
Sydney-Berkeley Driving Team, 274
System
context, 262, 265
engineering, 245, 252
runtime faults
scenarios, 262

T
Tartan racing, 20–21, 24, 27, 32, 39, 119
urban challenge system, 21, 24
TCP/IP, 32–33
socket, 25, 32

Team
AnnieWay, 12
CarOLO, 8, 13, 210, 250
Cornell, 13
Oshkosh Truck, 14
Sydney, 300
Track A, 7, 13–14, 249
Track B, 7, 13–14, 249
UCF, 14, 134
VictorTango, 14
TeamBots, 23
Technische Universität Braunschweig, 13, 210
Testing, 20, 29–30, 38, 45, 60–61, 63–65, 88, 94, 117, 126, 186, 188–200, 206, 244, 246–248
acceptance, 266
automated, 244, 246–247, 257
black box, 259
coverage, 255
CxxTest, 254
hardware-in-the-loop, 248
human in the loop, 129
regression, 299
track, 61–62, 75, 204, 293, 295–296, 299
virtual, 260
Texas, San Antonio, 8, 250
TNO PreScan, 268
TORC Technologies, 94
JAUS Toolkit, 117
Touch Panel, 116
Toyota, 300
highlander, 68
prius, 13
RAV4, 274–277
Trac, 252, 254–255
Trajectory, 111–112, 124, 134–137, 139, 141, 143, 152, 157–158, 145, 159–160, 162–163, 168, 172, 179–181, 184–186, 188–189, 191, 211, 214, 234, 236, 261, 279, 286–287, 290–291, 293
boundary conditions, 139, 149
clothoid-based, 158
collision free, 135, 144–145, 148, 151
constructive polylines, 155, 162, 170–172, 175, 177–180, 188–189
curvature, 156–170, 182-185, 189–190, 286
drivable, 157–158, 180, 182, 189
feasible, 134, 136–137, 141–142, 145, 147–149, 156, 190
kinematic model, 136, 138
metric, 182, 183, 185
open-loop, 291
optimal, 111, 142, 145, 147–149, 151
parameterized, 151
pearls, 230
polynomial model, 138–139, 141, 154
real-time, 145
search, 112
sharpness, 157–162, 164, 166–167, 182–183
smoothing, 299
valid, 190
Tsukuba laboratories, 210
Tyndall Air Force Base, 89

U
Universität
der Bundeswehr München, 12
karlsruhe, 12
University of
California, Berkeley, 266–267
Central Florida, 14
Florida, 89
Louisiana, Lafayette, 206
Pennsylvania, 13
Sydney, 274, 299
Technology, Sydney, 274, 299
Unix, 32, 38, 254
User interface, 21–23, 26–31, 33–35

V
Valgrind, 256
Vehicle
acceleration, 114, 215, 292
accelerator, 64, 234, 277, 279
actuation, 113–114, 276–277
brakes, 278
center of gravity, 202–203, 206
collision, 64, 237
design, 276
drive-by-wire, 96, 199, 276
driver assistance, 244, 269
dynamics, 114–115, 198, 268
electrical system
emergency flashers, 115, 123
engine control unit, 276
engine start, 200
failures, 294
hardware, 96, 276
headlights, 200
horn, 111
human control, 276
kinematics, 85, 121, 137, 141
orientation, 117–118, 267

V (*cont.*)
reaction time, 119, 123
self-driving, 210, 268
shifter, 87
speed, 86–87
stability, 102, 207
steering, *see* steering
transmission, 68
turn signal, 111, 116–117, 123, 198, 200
turning radius, 167, 169
wheel speed, 103, 241
windshield wipers, 200
VictorTango, 94
Virginia Tech, 7, 241
Vision, 68, 78–79, 134, 196, 274, 286
perception, *see* perception
stereo, 198–199, 299
Volkswagen Passat, 12–14, 212, 250, 268–269
Voting, 51, 58

W
Waypoint, 7, 9, 50, 59, 68, 71–73, 84, 95, 105–106, 119, 134, 141–142, 156–157, 160, 162, 168, 171, 179, 182–190, 199, 205, 215, 267, 282, 286–287, 291–293, 301–304
definition, 302
entrance, 106, 119
exit, 73, 83, 95, 105–106, 119, 122, 285–286, 301–303

Z
ZeroC, 284, 300
IceGrid, 284
ZFS, 254